This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2025.3630754

Optimizing VM Performance Monitoring on Commodity x86 Platforms with PMU Passthrough

Anonymous Authors

***Abstract*—Modern processors expose hardware performance monitoring capabilities through Performance Monitoring Units (PMUs). Profiling tools leverage these PMU facilities to analyze program execution. As the computation is increasingly shifting to virtual machines (VMs), commodity hypervisors like KVM have implemented virtual PMUs (vPMUs) to expose these capabilities to VMs. However, we found that KVM's current vPMU implementation introduces substantial overhead, leading to inaccurate measurements. This overhead stems from frequent VM exits caused by PMU access and sampling interrupts, as well as the complexity of emulation. To address this, we develop Direct vPMU (D-vPMU), a novel set of passthrough mechanisms for Intel x86 platforms running KVM. D-vPMU enables VMs to access the physical PMU and handle interrupts directly, thereby eliminating the virtualization overhead that currently plagues vPMU solutions. D-vPMU delivers better profiling accuracy and performance while strictly maintaining the security and isolation boundaries between guest and host PMU contexts.**

I. Introduction

Modern processors expose hardware performance monitoring capabilities through Performance Monitoring Units (PMUs) [32], [33] – specialized hardware components that record and report microarchitectural events during runtime. These PMUs enable profiling frameworks to analyze program execution characteristics at the hardware level. For instance, PMUs log events such as last-level cache misses, branch mispredictions, or the target addresses of the processor's most recently executed branch instructions. Software tools such as the Linux perf [16] and Intel Vtune [17] have been developed to facilitate configuring the PMU hardware and retrieving architectural profiling information. Developers [2] can utilize these tools to gather valuable insights about program execution and identify performance bottlenecks or errors in programs.

Organizations increasingly migrate workloads from on-premises infrastructure to virtual machines (VMs) running in the cloud. Major vendors, like Google and Amazon, adopt a commodity Linux KVM [31] hypervisor to host VMs on their server hardware. These hypervisors have evolved to virtualize essential processor features, including performance monitoring hardware, to enable seamless VM adoption across diverse computing environments. Hypervisors implement virtual PMU (vPMU) interfaces to expose hardware monitoring capabilities to guest VMs. KVM's implementation for the predominant Intel x86-based platforms utilizes this approach to implement PMU virtualization.

While this approach enables guest operating systems to leverages PMU features, it introduces substantial runtime overhead during profiling operations, compromising measurement accuracy and fidelity of performance characteristics. We identified that the overhead primarily stems from two factors.

First, to mediate and emulate accesses to PMU's Model-Specific Registers (MSRs), KVM employs a trap-and-emulate approach to intercept on VM's access to PMU MSRs. This triggers frequent VM exits. Each exit involves costly context switches between the VM and hypervisor execution contexts. Further, KVM employs complex emulation logic to properly isolate physical PMU resources between VMs and the host. The cumulative impact significantly degrades overall system performance, particularly in profiling scenarios.

Furthermore, KVM's approach to virtualize Performance Monitoring Interrupts (PMIs), the interrupts generated by the PMU, introduces significant latency in interrupt delivery to VMs. This latency is particularly detrimental to the accuracy of performance measurements in virtualized environments. PMIs serve critical functions in interrupt-based sampling and counter management, as they notify the processor when an event counter overflows. When overflow occurs, hardware automatically resets the counter, requiring monitoring software to quickly retrieve counter values to maintain measurement fidelity. However, KVM's indirect interrupt handling mechanism substantially delays this time-sensitive operation.

This paper proposes novel solutions to optimize the PMU virtualization overhead in VMs on commodity KVM Intel x86-based settings and enhance performance monitoring capabilities. Our solution, *Direct vPMU* (D-vPMU), incorporates passthrough mechanisms that optimize PMU virtualization, substantially reducing the cost of emulating PMU accesses and PMI delivery latency.

D-vPMU eliminates the trap-and-emulate overhead of accessing PMU's MSRs. This mechanism enables VMs to configure the PMU and retrieve the PMU's profiling event states directly from the hardware, without requiring hypervisor mediation, thereby accelerating profiling operations. Unlike the previous solution [8], D-vPMU multiplexes the PMU hardware between the hypervisor host and VMs to retain strict isolation between their PMU contexts, preserving security and profiling functionality for both guest and host environments.

As shown in Figure 1, a straightforward approach to managing PMU contexts would be implementing context switches at every VM exit, similar to methodologies employed by mediated passthrough vPMU [29], [34], which also proposed a PMU passthrough solution. However, this incurs significant overhead in each VM exit. To address this limitation, D-vPMU incorporates a more efficient *deferral* context switch strategy (see Figure 1). The idea is to perform context switches only when absolutely necessary rather than at each VM exit.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2025.3630754

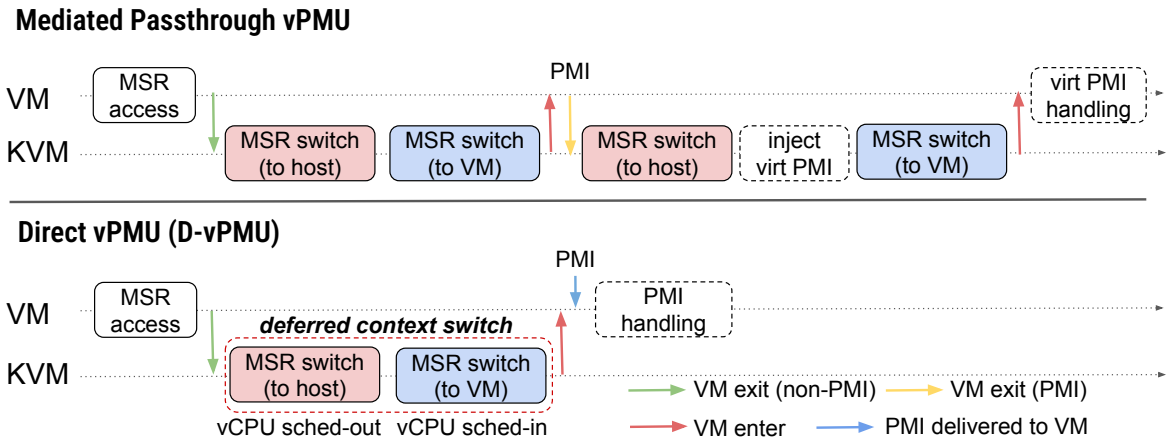

Fig. 1: VM execution comparison: Mediated Passthrough vPMU versus Direct vPMU

We exploit an observation that host PMU contexts typically remain inactive until virtual machine de-scheduling occurs. By strategically deferring PMU context switch operations until a VM's vCPU de-scheduling, D-vPMU substantially reduces virtualization overhead by eliminating context switches in critical VM exit paths, while strictly isolating guest and host PMU states.

Further, D-vPMU introduces a PMI passthrough mechanism that securely enables direct PMI delivery to VMs. This eliminates VM exits triggered by frequent sampling events and virtual PMI injection required in prior approaches, including KVM and mediated vPMU passthrough. As shown in Figure 1, with PMI passthrough, VMs are allowed to handle PMIs without requiring KVM's intervention. D-vPMU ensures efficient interrupt delivery to the intended recipient, whether VM or host, preserving functionality and critical security boundaries, guaranteeing that VMs cannot intercept interrupt signals intended for the host system.

We extended KVM in the mainline Linux v6.9 to incorporate D-vPMU's passthrough optimizations. Evaluation of our prototype revealed that D-vPMU outperforms the state-of-the-art solution [29], [34], by employing deferral PMU MSR context switch and PMI passthrough, and delivers substantial performance gains to KVM: D-vPMU reduced the cost of micro-level PMU access operations more than $20x$ while improving the performance of VM profiling with intensive PMI deliveries by more than $7x$. To validate D-vPMU's efficacy, we conducted real-world tests using the Linux perf [16] tool within a VM to profile an Nginx server. Results demonstrate that D-vPMU significantly reduces sampling overhead, resulting in a modest profiling overhead to Nginx and achieving a $10x$ improvement in virtualization overhead. We also demonstrate that D-vPMU achieves significantly less sample loss in active profiling scenarios. These enhancements enable precise capture of program execution behaviors in VM environments.

Security researchers commonly leverage the isolation properties of VMs for software testing, extending hypervisors to create fuzzing frameworks. We anticipate these frameworks will significantly benefit from the superior profiling capabilities enabled by D-vPMU, allowing for more informed fuzzing decisions, ultimately facilitating vulnerability discovery. We ported D-vPMU to a KVM-based fuzzing framework [23]. Our evaluation demonstrates that fuzzers can now leverage rich hardware profiling information for future enhancements without significant throughput degradation. This underscores the practical value of our work, not only for cloud deployments but also crucially for in-house development and testing scenarios where performance efficiency is often prioritized.

In summary, this paper makes the following contributions.

- We identified 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.
- We introduced D-vPMU, which incorporates MSR and PMI passthrough optimizations to enhance performance while retaining functionality, isolation, and security.
- We conducted thorough evaluations of D-vPMU and demonstrated its efficacy compared to existing solutions.

II. Background

A. Intel VMX and KVM

Intel introduced Virtual Machine Extension (VMX) to support unmodified virtual machines. Intel VMX incorporates an in-memory data structure called the Virtual Machine Control Structure (VMCS). VMCS contains the CPU register states of the hypervisor host and VM in the host and VM state area. VMCS manages VM entry/exit conditions and keeps the CPU state during transitions between host and guest execution.

KVM [31] integrates a host Linux kernel to support virtual machines. When a VM is created, KVM establishes a dedicated thread for each virtual CPU (vCPU) requested, allowing these threads to be scheduled on physical CPUs like any other process. KVM leverages Intel VMX and creates a VMCS for each vCPU. When KVM enters the VM, VMX saves KVM'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. KVM configures VMCS's VM control fields to configure the VM's behaviors.

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2025.3630754

For instance, KVM can allow a VM to execute instructions to access MSRs without VM exits.

B. Intel Performance Monitoring Unit (PMU) and Last Branch Record (LBR)

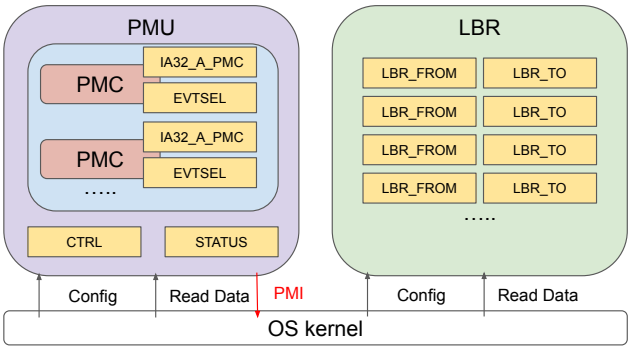

Fig. 2: Intel PMU and LBR

Intel PMU is a per-CPU-core module with several MSRs and performance monitoring counters (PMCs). The latter counts the core's architectural events. The software uses the RDMSR and WRMSR instructions to read from and write to the MSRs. As shown in Figure 2, the PMU includes two MSRs: IA32_PERF_GLOBAL_CTRL (referred to as ***CTRL***) and IA32_PERF_GLOBAL_STATUS (referred to ***STATUS***), respectively, for software to enable/disable and get the status of each PMC. A PMC 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 exceeds the bit width value. On the other hand, the OS kernel 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.

Performance Monitoring Interrupt (PMI). The software programs EVTSEL to raise a PMI 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. Intel processors deliver the PMI by default as a non-maskable interrupt (NMI). Therefore, PMIs cannot be masked by clearing the interrupt flag (IF) of the RFLAGS register (the per-CPU register that holds various status and control bits); instead, the OS kernel can configure the local Advanced Programmable Interrupt Controller (APIC) to mask and unmask PMIs. If the hardware is configured to generate PMIs, it sends a signal to the local APIC to inform that a PMI should be delivered. When the local APIC receives the signal, it first checks whether the PMI is masked (otherwise the PMI is dropped). 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.

The **Intel Last Branch Records (LBR)** [39] is a per-CPU-core module that logs profiling information, such as the source and destination addresses of recently taken branches. LBR entries are limited and can be overwritten, leading to data loss. To mitigate this, a PMC can be configured to generate PMIs at regular intervals before LBR's capacity is reached, allowing the OS kernel to read and preserve LBR records before they are replaced.

C. Performance Counters for Linux

The Performance Counters for Linux (PCL) is a Linux subsystem that abstracts hardware profiling features (i.e., PMU and LBR) into a unified structure (*perf_event*). It provides a comprehensive framework for performance analysis, including the perf_event_open() system call, a PMI handler, and support for context-switching PMU hardware. Software in Linux must utilize the profiling hardware through PCL, as direct access can lead to incorrect PMI handling and improper PMU state management during context switches, resulting in inaccurate profiling data. This PCL abstraction ensures consistent and reliable performance monitoring across various hardware and software contexts in Linux systems.

D. PMU Virtualization in KVM

Emulating PMU MSRs. KVM uses trap-and-emulate to virtualize VMs' PMU accesses. It configures the VMCS to trap VMs' MSR readings and writings in the profiling hardware so that KVM can emulate them. KVM maintains a virtualized abstraction, vPMU, 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 and routing the operation to the vPMU. When a VM enables its virtual PMU, KVM registers a corresponding *perf_event* with the host PCL. KVM derives the configuration of the perf_event based on the states of the virtual hardware. For instance, KVM obtains the target hardware event and privilege level from the virtual EVTSEL. The PCL stores the counter values retrieved from the PMC on the hardware to perf_event. When the VM later reads from IA32_A_PMC to get the counter value, KVM retrieves the value from the associated perf_event and returns the value to the VM. In Linux, the PCL frequently reads and writes to PMU's MSRs. Consequently, profiling with PCL in VMs can be much slower on KVMs than on bare metal.

PMI handling in KVM. KVM configures the hardware to route PMIs to itself during VM execution. A PMI causes a VM exit. KVM subsequently invokes the PMI handler from the host PCL to handle the PMI. If the VM should receive the interrupt, the PCL later notifies KVM, which injects a virtual PMI into the VM, leading to the invocation of the guest PCL's PMI handler. This multi-layered process significantly adds to PMI delivery latency.

E. Non-Maskable Interrupt (NMI)

An NMI is a type of interrupt that users cannot disable using the processor's interrupt masking mechanisms (e.g., set the interrupt disable flag). Linux uses NMIs to support critical system management and diagnostic functions. Hardware and software can both generate NMIs. Aside from PMIs, hardware-generated NMIs can come from watchdog timers, ensuring system responsiveness. The software initiates NMIs by configuring the local APIC to send inter-processor interrupts (IPIs)

with NMI delivery mode. Linux leverages this mechanism for various purposes, such as requesting a stack trace dump.

Linux NMI handling. Upon receiving an NMI, Linux traverses the list of registered NMI handlers. Due to the varied sources of NMIs, each interrupt arrives with essential ***contextual information***. This accompanying information is critical for appropriate NMI handling. KVM ensures that the host's *contextual information* is inaccessible to VMs. This information resides in the memory of KVM's Linux host kernel or hardware. Each NMI handler examines the contextual information to determine the interrupt's purpose before executing its specific logic. Linux marks an NMI as unknown if no registered handler claims responsibility. NMI handlers clear associated information upon finishing execution.

To illustrate how contextual information is used, we present a case study examining a few NMI handlers in the Linux kernel, focusing on their associated information checks.

- The PMI handler checks a global counter maintained by PCL and PMU's STATUS register. The counter records the number of registered performance monitoring events in the kernel, and STATUS shows which PMCs overflow. If no registered performance monitoring events are in the kernel, or STATUS indicates no PMC overflows, the PMI handler would skip the actual handling logic.
- The NMI handlers that dump stack trace and stop CPU's execution both check a global bitmap maintained by the Linux kernel. Before a kernel thread sends NMI to target processors, it updates the bitmap to set only bits associated with target processors. Upon receiving an NMI, the NMI handlers dump the stack trace and stop the CPU's execution, checking if the corresponding bit of the current processor is set in the bitmap. If not, they skip the underlying logic.

NMI blocking and PMI masking. Intel processors employ hardware mechanisms to prevent nested NMI and PMI handling [36]. Upon receiving an NMI, the processor blocks subsequent NMIs until an interrupt return (IRET) instruction is executed. Linux ensures atomic NMI handling due to the critical nature of these interrupts.

During NMI handling, Linux disables preemption and interrupts until all handlers are complete. It executes IRET when returning from all handlers and disables NMI blocking, allowing subsequent NMIs to be processed. Linux actively blocks NMIs only within NMI handlers and disables the blocking elsewhere. Similarly, after raising one to the processor, the local APIC masks future PMIs. Linux's PMI handler unmasks PMIs upon completion.

F. Motivating Example

Figure 3 illustrates the workflow of a motivating example: a VM on KVM registers a sampling event whose target is branch instructions in rings 1-3 with a sampling period of 10,000.

Initially, the guest process registers a performance monitoring event to guest PCL via perf_event_open(). Guest PCL parses the first argument (struct perf_event_attr) and attempts to write EVTSEL to 0x1100c4 to (1) configure PMC to count branch instructions in ring 1-3, and (2) configure PMC to generate a PMI on overflow, which causes VM exit and KVM

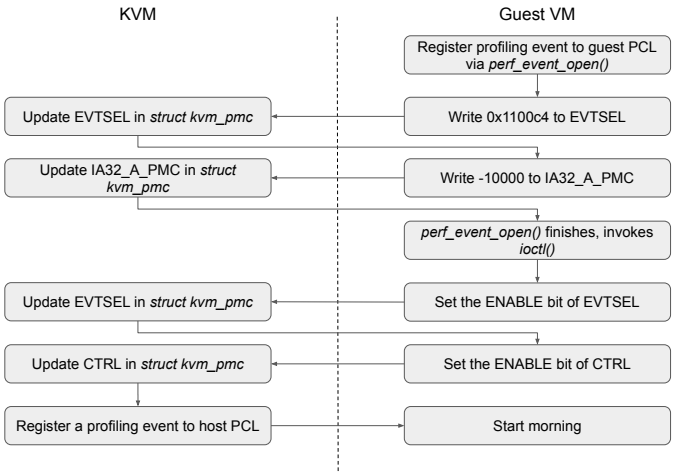

Fig. 3: Workflow of registering and starting a sampling event on KVM

updates EVTSEL in struct kvm_pmc in the vPMU. Then, the guest PCL attempts to write IA32_A_PMC to -10,000 so that the PMC can overflow after 10,000 branch instructions are executed. This causes the VM to exit again, and KVM updates IA32_A_PMC in struct kvm_pmc in vPMU.

After perf_event_open() finishes, the guest makes an ioctl() to start the sampling event. The ioctl handler from the guest Linux first attempts to set the ENABLE bit in EVTSEL, which causes a VM exit and KVM updates EVTSEL in struct kvm_pmc in the vPMU. Then, the ioctl handler attempts to set the ENABLE bit in CTRL, which causes a VM exit and KVM updates CTRL in the vPMU. Later, KVM observes that both ENABLE bits are set in EVTSEL and CTRL and starts inferring the attributes of the performance monitoring event. The value 0x1100c4 of EVTSEL indicates a sampling event, and the target hardware event is branch instructions executed in rings 1-3. Eventually, KVM registers a performance monitoring event with the host PCL according to the inferred attributes, and the PMC starts working after VM entry.

After 10,000 branch instructions are executed, the PMC overflows and generates a PMI (delivered as NMI). Figure 4 shows the workflow of PMI handling in a VM. For simplicity, we ignore VM exits due to MSR accesses in the figure. When the processor receives an NMI in VMX non-root operation, a VM exit occurs, and KVM invokes NMI handlers in the host, which eventually invoke the PMI handler of the host PCL. KVM and the PCL function asynchronously. KVM registers a performance monitoring event in the host PCL. The PMI handler of the respective host PCL reads the hardware's IA32_A_PMC and updates the statistics of the registered event, then invokes KVM's callback function, which is provided by KVM and injects an NMI into the VM.

On a subsequent VM entry, the guest's PMI handler is invoked to read hardware IA32_A_PMC, which causes the VM to exit again. KVM handles the exit by fetching the value from struct kvm_pmc and returns the value to the guest. Eventually, the execution of the guest process continues after the PMI handler of the guest PCL finishes.

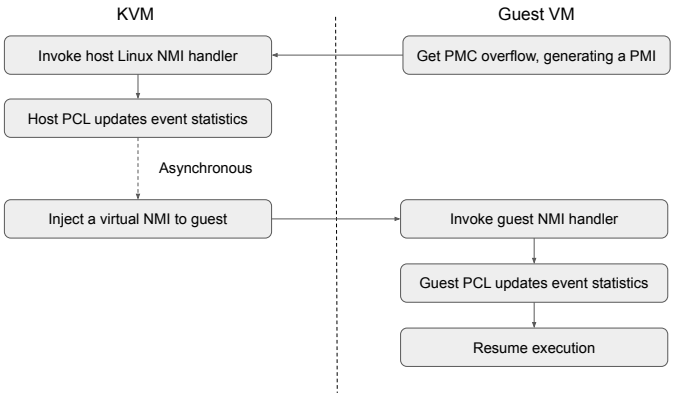

Fig. 4: Workflow of PMI handling on KVM

III. Design

In this work, we introduce D-vPMU, which incorporates two novel mechanisms, *MSR passthrough*, and *PMI passthrough*, to eliminate VM exits incurred by PMU MSR accesses and PMI handling, to optimize the performance of KVM's PMU virtualization while preserving the security and functionalities of the host and guest OS kernel.

A. MSR passthrough

D-vPMU introduces MSR passthrough to optimize PMU virtualization. It configures the VMCS to allow VMs to access the hardware's PMU MSRs directly. This removes all VM exits shown in Figure 3 on KVM. Specifically, D-vPMU clears corresponding bits for PMU MSRs in the MSR bitmap in VMCS to disable VM exits caused by VMs executing RDMSR/WRMSR against these MSRs. This eliminates the trapping and KVM's MSR emulation overhead for managing vPMU states and utilizing the host PCL (e.g., monitoring event registration), allowing the guest PCL to read and write PMU MSRs natively. We carefully design the MSR passthrough mechanism to achieve the following two properties:

- **MSR-P1**: Isolate host and VM MSR states.
- **MSR-P2**: Achieve **MSR-P1** with modest overhead.

*1) **Property MSR-P1**:* KVM's original implementation relies on the host's PCL to facilitate PMU virtualization. KVM registers performance monitoring events for each VM. This allows KVM to utilize PCL's existing features that multiplex hardware PMU states between contexts of each different thread on Linux. Specifically, the host PCL saves and restores PMU MSR states on the hardware when the host and vCPU thread context switches. When utilizing MSR passthrough, the host PCL is detached from the VM. To multiplex the hardware's PMU resources between the VM and host, D-vPMU extends KVM to perform context switching of the hardware PMU MSRs between host and VM execution contexts.

*2) **Property MSR-P2**:* To realize **MSR-P1**, host and VM PMU MSRs can be context-switched upon VM exit. Previous work [29], [34] implemented this approach by configuring the VMCS for automatic MSR state context-switching on each VM exit. Specifically, 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 Intel hardware, more than 10 relevant MSRs must be saved in a VM exit and restored in a VM enter. We found that this VMCS-based approach could induce significant performance overhead.

To ensure **MSR-P2**, D-vPMU introduces a *deferral* PMU MSR context switch strategy. D-vPMU leverages Linux's preempt notifier framework, which allows subsystems to register callbacks that execute before and after a task is preempted or switched to another CPU. KVM already utilizes this notifier framework to efficiently manage guest-to-host transitions when virtualizing hardware resources. The approach enhances performance by optionally context-switching VM states when necessary. When a VM is about to be preempted, KVM's preempt notifier *sched_out* callback saves virtual CPU state and hardware register values. Conversely, when the VM is scheduled back, the *sched_in* callback restores the saved state to the hardware.

To support the deferral approach for PMU MSRs, we register callback functions to Linux's preempt notifier to context switch MSRs whenever Linux schedules in or out of a VM. 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, our approach context switches MSRs only when the vCPU does not use the states or when the vCPU gets executed, thus resulting in much better performance. Figure 1 depicts this mechanism in detail.

Our approach defers context switching of all PMU MSRs except for the CTRL MSR. We configure the VMCS to save and restore the CTRL MSR on VM exits and entries, specifically to isolate the PMU usage between the VM and the host. This prevents hardware events within the hypervisor from affecting the PMU state (i.e., IA32_A_PMC) accessed by the VM. If all MSR saves and restores were deferred, the PMU would continue counting and recording after a VM exit. To prevent this, we disable the PMU upon a VM exit, immediately after the VMCS performs the context switch of CTRL. This guarantees that instructions executed in KVM do not influence the VM's PMU state.

B. PMI passthrough

As mentioned in subsection II-E, PMIs are delivered as NMIs on Intel hardware. KVM virtualizes PMIs by requiring VM exits and applying complex handling logic (see Figure 4) to inject virtual PMIs. D-vPMU introduces a PMI passthrough approach to eliminate PMI delivery overhead, enabling guests to handle PMIs directly. The challenge is that VMCS does not support fine-grained control of NMI trapping – Intel VMX only allows the hypervisor to disable VM exits of all NMIs.

To support PMI passthrough, D-vPMU disables NMI exiting in the VMCS and allows VMs to handle PMIs without exiting, avoiding PMI delivery delays. Although this approach is feasible and effective for in-house testing, it poses unique security

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2025.3630754

challenges to external deployments; we must ensure that a VM's NMI handling does not affect the host or other VMs. We detail how PMI passthrough builds on the observed insights from Linux's NMI handling methodology (see subsection II-E) to effectively enforce the two listed critical properties to retain the safety and correctness of KVM's NMI handling.

- **PMI-P1**: NMIs are handled by the intended entity.
- **PMI-P2**: Isolate NMI/PMI blacking/masking status.

*1) **Property PMI-P1**:* Since NMIs signal critical events, we must ensure NMIs are handled by the intended entity (e.g., the host or VMs) so that passing them through to VMs incurs no functionality and security issues. In our discussion, we consider two scenarios: (1) host NMIs sent to VMs and (2) VM NMIs sent to the host.

Host NMIs sent to VMs. NMIs used by the KVM host may be misrouted to VMs. By default, VMs cannot recognize host NMIs. The guest kernel that receives a host NMI sees them as unknown. We created two mechanisms (*hypercall* and *monitoring*) to safely pass through PMIs in two scenarios: cooperative and non-cooperative VMs.

For cooperative VMs, we created a *hypercall mechanism*, allowing VMs to notify KVM about unknown NMIs. Upon receiving this hypercall, KVM redirects the NMI to the host kernel's NMI handlers for proper processing. This approach ensures that misrouted NMIs are correctly handled without compromising legitimate purposes. For cloud deployments, we expect vendors to provide guest kernel-prebuilts that are extended to employ the hypercall to cooperate with KVM.

However, non-cooperative VMs may reject using hypercalls to report unknown NMIs and hide these NMI from the host. To address the security challenge, we extended KVM to install an ***NMI monitor*** that is invoked to check NMI's *contextual information* to detect signaled NMIs at each VM exit. We exploit a key observation made in this work that the VM could not access the ***contextual information*** stored in the host's memory or on hardware (see subsection II-E). A VM's attempts to access the contextual information are intercepted by KVM supporting D-vPMU. This forbids a rogue VM from accessing or tampering with the information to compromise the host's NMI handling.

VM NMIs sent to the host. Performance Event Skid [28] is the delay between a performance event's occurrence and delivery. When passing NMIs through to VMs, this skid can cause PMIs intended for VMs to be mistakenly received by KVM. This happens when VM exits occur during the skid period, such as when an interrupt arrives at the processor. To address this misroute, we install a PMI handler in KVM's host Linux. When the host receives an unexpected PMI, this handler sets a flag instructing KVM to inject the PMI into the VM upon the next VM entry, ensuring proper routing of PMIs in the presence of performance event skids.

*2) **Property PMI-P2**:* Figure 5 shows that the PMI handler in a VM masks PMI and blocks NMI upon receiving an NMI. Unlike the native environment, VMs cannot handle NMIs/PMIs atomically. When a VM's NMI handler executes, VM exits can occur due to external events like timer interrupts. In this scenario, the trapping prolongs the duration during which NMIs and PMIs remain blocked and masked on the processor, causing potential NMI/PMI loss.

D-vPMU enforces **PMI-P2** to isolate the interrupt blocking or masking status of a VM on the hardware from affecting the host or other VMs' execution. As shown in Figure 6, D-vPMU ensures that, upon VM exit, checks the *blocking by NMI* bit in the VMCS to detect active NMI blocking. When this bit is set, it executes D-vPMU's *NMI unblock helper* in KVM outside the interrupt handler. This helper utilizes a carefully crafted fake interrupt stack to execute *IRET* as a *NOOP*, effectively unblocking NMI without affecting other processor states. NMI blocking status is automatically restored on VM entry by reconfiguring the VMCS. D-vPMU ensures consistent NMI blocking status before and after VM exits, maintaining the integrity of the system's interrupt handling capabilities.

```
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

Further, similar to the issue of NMI blocking, when employing PMI passthrough, interrupting the VM's PMI handling process delays unmasking PMIs and compromises functionalities that depend on PMIs. To resolve the issue, as shown in Figure 6, we manually unmask PMI when a VM (i.e., its vCPU thread) is scheduled out on a processor, ensuring that PMI masking does not affect other tasks on the host. We also record whether PMI was masked before the VM was scheduled out. If it was masked, we re-mask PMI when the vCPU thread is scheduled back in.

Passing PMIs directly to VMs requires ensuring that VMs can unmask them to receive subsequent interrupts. However, KVM prevents VMs from programming the hardware's local APIC to unmask PMIs. To solve this, we extended KVM to trap VM updates to the local APIC and unmask PMIs on the hardware on the VM's behalf when passthrough is enabled.

IV. Evaluation

We added and modified 465 lines of C code to KVM in the mainline Linux v6.9 to implement the proposed MSR and PMI passthrough to support D-vPMU. We use the following experimental setup to evaluate the performance of bare-metal and VM environments. Our evaluation was conducted on machines with dual Intel Xeon Silver 4114 10-core CPUs (2.20GHz) and 192 GB DDR4 RAM. For bare-metal (denoted **BM**) tests, we employ Ubuntu 20.04 with Linux kernel v5.5.0. For VM tests, we employ three configurations: (1) KVM in

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2025.3630754

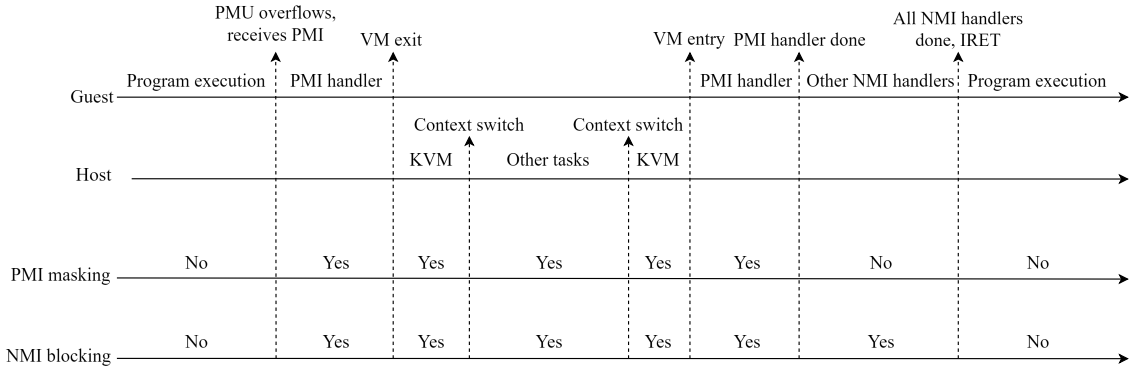

Fig. 5: Original timeline of NMI blocking and PMI masking with passthrough

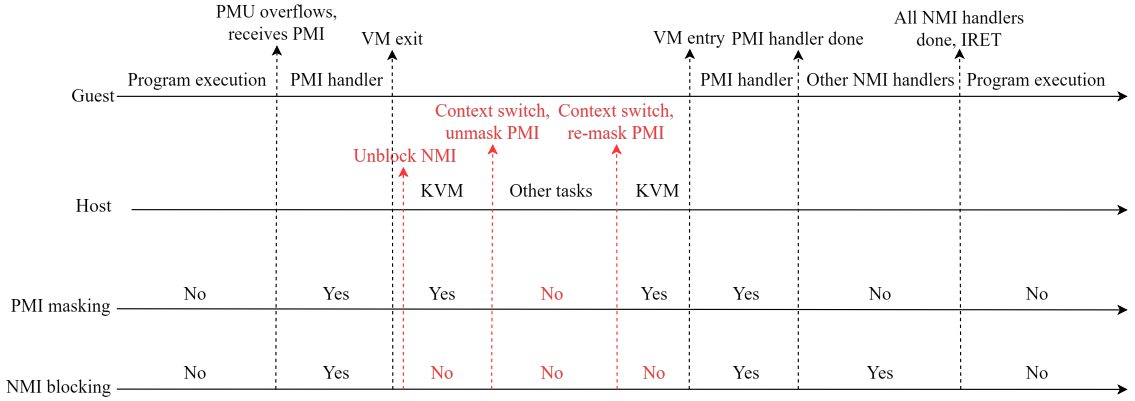

Fig. 6: Timeline of NMI blocking and PMI masking with passthrough after securing P2

mainline Linux v6.9, (2) KVM with D-vPMU extensions, and (3) KVM in Linux v6.7 with mediated passthrough vPMU support. For (3), we used the Linux version provided by the authors in their RFC release [27]. We denote the results respectively for (1), (2), and (3), as **KVM**, **D-vPMU**, and **Mediated**. All VMs use Ubuntu 20.04 with Linux v5.5.0. We use the same configuration to compile all host and guest Linux to unify the test environment to the best of our efforts. We used Ubuntu 22.04 on all hosts. All VMs were launched using QEMU v8.0.0. We allocated two vCPUs for the evaluation described in subsubsection IV-B1 and one vCPU for other tests, with each VM assigned 4GB of memory.

We conducted the following experiments to comprehensively evaluate the efficacy of D-vPMU compared to the state-of-the-art solutions. We first measured micro-level performance in subsection IV-A. Like previous works [37], [38], we extended the KVM-unit-test framework to evaluate low-level operations and measure the impact of D-vPMU on their performance. Additionally, we created a custom benchmark that generates various PMIs during VM execution to quantify VM performance PMI handling. Next, in subsection IV-B, we evaluated the performance of real-world profiling scenarios. They include a user employing the widely used Linux perf tool [16] to evaluate common VM workloads, such as profiling a Nginx web server. We also ported D-vPMU to a prior KVM-based fuzzing framework [23] to test the performance of VM profiling that acquires LBR information during fuzzing kernel drivers. Finally, we evaluated the accuracy of VM profiling results in subsection IV-C in various scenarios, including those where multiple VMs, as well as host and VM profiling, are performed simultaneously using Linux perf.

A. Performance Results

1) MSR Accesses: We first evaluated the elapsed cycles for executing a single RDMSR/WRMSR instruction to access PMU and LBR MSRs on a VM and bare metal. We extended KVM-unit-tests [26] to obtain elapsed cycles for RDMSR/WRMSR in a VM. Regarding bare metal (BM), we executed instructions identical to those in KVM-unit-tests in a custom system call. The result is shown in Table I.

Table I (a) shows KVM incurs significant overhead for VM's PMU MSR access due to trap-and-emulate. In contrast, D-vPMU and mediated vPMU allow native MSR access and offer similar optimized performance. Compared to the original KVM implementation, D-vPMU achieves more than 20x enhancement in MSR read operations and 14x improvement in MSR write operations. The only exception is that D-vPMU and mediated vPMU introduce overhead by trapping EVTSEL register accesses for security reasons to filter out sensitive PMC accesses, preventing guests from acquiring hardware information. D-vPMU incurs much less overhead than mediated vPMU because of deferring context switching PMU MSRs. This demonstrates its effectiveness in reducing

This article has been accepted for publication in IEEE Transactions on Cloud Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCC.2025.3630754

TABLE I: Consumed CPU cycles. Left: MSR accesses (RO = read-only MSR). Right: VM exit instructions.

(a) MSR Access Overhead

MSR	KVM		D-vPMU		Mediated		BM	
	R	W	R	W	R	W	R	W
EVTSEL	3,134	3,200	4,158	4,197	9,405	9,401	107	186
IA32_A_PMC	3,128	3,202	142	225	142	225	90	175
CTRL	3,076	3,134	138	184	139	185	87	134
STATUS	3,120	RO	138	RO	139	RO	87	RO

(b) VM Exit Instruction Overhead

Instruction	KVM	D-vPMU	Mediated
CPUID	2,981	4,007	9,286
VMCALL	2,951	3,950	9,196
INL	3,221	4,225	9,462
OUTL	3,231	4,232	9,472

the MSR access overhead, enforcing the property **MSR-P2** (see subsection III-B).

2) VM Exit Overhead: D-vPMU installs an NMI monitor that checks the contextual information to safely enable PMI passthrough (see subsection III-B). To measure the overhead resulting from this mechanism, we evaluated microbenchmarks from the KVM-unit-tests [26]. We present the results in Table I (b). As shown, D-vPMU incurs an average of roughly 1,000 cycles of overhead for every benchmark compared to KVM. The check requires roughly 230 cycles, while additional 800 cycle overhead is needed for context switching EVTSEL MSRs. The overhead is imposed on each VM exit and added to operations, such as the CPUID instruction or VMCALLs, that are irrelevant to PMI delivery. However, we believe the trade-off is justified by the substantial PMU performance gains achieved. Table I (b) shows that, compared to D-vPMU, mediated vPMU passthrough, resulted in approximately 3× higher overhead than KVM. Context switching MSRs on every VM exit causes a significant slowdown. D-vPMU employs a deferral MSR switching approach to avoid such overhead.

3) Profiling Under Varying Numbers of PMIs: In addition, we measured the execution time of a simple program that uses PCL in VM and on bare metal to show that MSR and PMI passthrough play a role in reducing overhead. Specifically, the program registers a performance monitoring event to count branch instructions in rings 1-3 and runs an empty for loop with N iterations (i.e., N branch instructions are executed). The event is configured to generate a PMI for every M branch instruction. Both M and N are configurable. The program is expected to receive $\frac{N}{M}$ PMIs during its execution. We fixed N to 100,000 and mutated M with different values. For instance, to measure the execution time of the program without PMIs and with 1,000 PMIs, we set M respectively to 200,000 and 100. The results are shown in Table II. To demonstrate the benefits of PMI passthrough, we analyze the results of D-vPMU with two configurations: D-vPMU with MSR passthrough only and with both MSR and PMI passthrough.

When no PMIs are sent, the execution time is approximately 0.2 ms in all cases. KVM's performance worsens as M increases. In the worst case, when 1,000 PMIs were issued, KVM is more than 10 times slower than bare metal. KVM's trap-and-emulate approach significantly increases profiling overhead when using PCL in VMs. Results show that D-vPMU reduces the virtualization overhead. D-vPMU incorporating PMI passthrough significantly reduces KVM's overhead by more than $7x$ when 1,000 PMIs were sent. Note that mediated vPMU does not enable direct PMI delivery, yielding similar performance to D-vPMU employing MSR passthrough only.

B. Real-World Usage

We conducted additional experiments to evaluate D-vPMU in real-world testing scenarios. We evaluated scenarios when a VM uses the Linux perf tool [16] to profile a dummy program and a server application, Nginx. In addition, we measured a novel application of PMU features by harnessing hardware-level profiling support to enhance software fuzzing.

```
#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 Dummy Program

1) Linux Perf Tool:

a) Simple Workload: We crafted a simple workload that executes empty loops with different iterations in different functions and profiled the workload with the Linux Perf Tool. The workload is shown in Listing 2. The execution time of each function is the number of iterations it executes, proportional to those executed by all its callees. For instance, function a() executes 2×10^8 iterations and calls function aa(), which executes 10^8 iterations. Given number of iterations in the workload is 10^9, the execution time that function a() accounts for is $\frac{10^8+2\times10^8}{10^9} = 30\%$. We used perf to sample the elapsed CPU cycles in rings 1-3 and captured stack traces for the workload, which we used to generate the execution time distribution (see Table III). We also measured the total execution time of the workload with and without profiling (presented in Table IV). The results show that all setups generate accurate profiling results. However, compared to BM, KVM and mediated vPMU bias the performance characteristics, causing a 20% and 15% cost longer execution time when profiling. In contrast, D-vPMU results in accurate results while achieving lower overhead.

b) Nginx: We measured the profiling overhead of VM application workloads. To mimic a practical setting, we ran Nginx in a VM to serve requests sent by wrk [25] on a bare metal client running on a separate but identical machine,

TABLE II: Execution Time of Program Under Profiling with Varying Numbers of PMIs

# of PMIs	KVM	D-vPMU MSR	D-vPMU MSR + PMI	Mediated	BM
0	0.236 ms	0.236 ms	0.236 ms	0.236 ms	0.195 ms
10	0.507 ms	0.365 ms	0.261 ms	0.362 ms	0.216 ms
100	2.929 ms	1.521 ms	0.583 ms	1.492 ms	0.393 ms
1000	27.380 ms	13.281 ms	3.802 ms	12.944 ms	2.145 ms

TABLE III: Profiled Time Distribution of the Dummy Program

	KVM	D-vPMU	Mediated	BM	Expected
a	31.82%	31.22%	30.28%	29.58%	30%
aa	9.72%	9.75%	10.14%	9.72%	10%
b	39.02%	38.99%	39.86%	39.59%	40%
bb	29.27%	29.24%	29.14%	29.62%	30%
bbb	9.76%	9.76%	9.71%	9.89%	10%
c	29.15%	29.77%	30.82%	29.85%	30%
main	100%	100%	100%	100%	100%

TABLE IV: Execution Time of the Dummy Program

	KVM	D-vPMU	Mediated	BM
Without Profiling	2.36 s	2.36 s	2.36 s	1.92 s
With Profiling	2.84 s	2.58 s	2.72 s	2.06 s
Overhead	20.3%	9.3%	15%	7.3%

connected to the server machine via 10 Gbps Intel X520-DA2 NICs. We configured the VM to use a virtio-net device. We profiled the VM's Nginx worker process using Linux perf, sampling CPU cycles (ring 1-3) and capturing stack traces for flame graph generation. The client wrk stress-tested the server for 30 seconds (1 thread, 16 connections) to test throughput.

Table V presents the results without profiling. D-vPMU and mediate vPMU, respectively, incur a 2.3% and 5.6% overhead compared to KVM in transferred data per second, due to the higher VM exit cost. However, as shown in Table VI, when under profiling, KVM performs much worse than BM, incurring a more than $2x$ higher overhead. This stems from frequent VM exits for MSR accesses and PMIs. The results suggest that profiling slows down Nginx and makes it unable to reflect the original performance, potentially obscuring the issues that developers intended to identify. Table VI shows that D-vPMU's passthrough optimizations significantly enhance performance over KVM when profiling the Nginx worker process in a VM. For requests per second, comparing profiling to no profiling, D-vPMU's 2.4% overhead (19,731.47/19,276) is $10x$ improvement to mediated vPMU's 24% (19,073.45/15,401.6) and $16x$ improvement to KVM's 38% (20,210.77/14,616.86) slowdown.

The results indicate that D-vPMU significantly reduces the profiling overhead, delivering better performance than the mediated vPMU, and enables reliable results that reflect performance characteristics in virtualized environments.

2) Fuzzing: Fuzzing is a software testing technique that provides random input data to a target program to identify crashes or memory leaks. Recently, coverage-guided fuzzing, which uses code coverage to inform input generation, has become the predominant approach in fuzzing research. Syzkaller [24] represents a state-of-the-art coverage-guided fuzzer targeting operating system kernels. Building on this, Agamotto [23] enhances kernel device driver fuzzing by moving the process into a VM and leveraging VM snapshots for capturing fuzzing progress to accelerate performance.

Branch coverage, defined as the ratio of taken branches to total branches, is a standard metric for code coverage. It provides insight into the extent of basic block execution. We propose that utilizing Intel's LBR can significantly enhance coverage-guided fuzzing for several reasons: First, LBR offers a nonintrusive alternative to instrumentation for collecting code coverage data. Instrumentation increases program size and may be infeasible when disk space is limited. LBR circumvents this issue by leveraging hardware-collected branch information. In addition, LBR emits the execution order of branch instructions, elapsed cycles, and misprediction records. Fuzzers can leverage the information to generate more targeted seeds to improve efficiency. For example, preemptively running the fuzzing process on the initial corpus with LBR enabled can reveal critical branches that potentially impede progress. This insight allows the fuzzer to strategically generate or mutate test cases, more efficiently navigating these challenging branches and accelerating the overall exploration of the target program's execution paths.

To exploit LBR's potential, we extended Agamotto [23] to use LBR profiling for all executed system calls. To minimize VM profiling overhead with LBR, we ported D-vPMU's MSR and PMI passthrough optimizations for Linux v6.9 to Linux v4.18.20 (the version that Agamotto uses). Table VII compares the number of executed test cases (system calls) within 3 hours for Agamotto, with and without LBR enabled, across seven kernel drivers. We envision a proprietary use case for Agamotto. Unlike the other experiments in this paper, we disable the event-selector trapping to maximize performance. Higher numbers indicate better performance. Note that we were unable to collect KVM's results as baseline. This is due to much slower virtualization performance; fuzzing cannot be completed on Agamotto when LBR profiling is activated.

We utilized PMU to generate a PMI every 32 branch instructions taken, thereby avoiding the overwriting of LBR buffers (with 32 entries) by subsequent branch instructions. We extended the guest kernel to expose the contents of the LBR buffer and store them in memory shared with KVM. We implemented a new ioctl() handler in KVM that allows Agamotto's fuzzer in QEMU to retrieve LBR information. We extended QEMU to invoke this ioctl() during VM execution, simulating a scenario where LBR data guides fuzzing decisions (we did not utilize it for actual fuzzing). As shown in Table VII, LBR profiling results in an average 22.3% reduction in the total number of tested system calls to [23]. We deem it a reasonable tradeoff to acquire LBR records to enhance the

TABLE V: Nginx throughput without profiling

	KVM	D-vPMU	Mediated	BM
Avg Latency	787.10 us	803.82 us	831.46 us	487.65 us
Max Latency	13.23 ms	14.09 ms	13.8 ms	4.98 ms
Requests/sec	20,210.77	19,731.47	19,073.45	31,302.15
Trans Data/sec	173.16 MB	169.05 MB	163.75 MB	268.16 MB

TABLE VI: Nginx throughput with Perf profiling

	KVM	D-vPMU	Mediated	BM
Avg Latency	1.1 ms	827.87 us	1.03 ms	500.38 us
Max Latency	13.56 ms	12.62 ms	20.09 ms	6.6 ms
Requests/sec	14,616.86	19,276	15,401.6	30,730.53
Trans Data/sec	125.23 MB	164.58 MB	131.95 MB	263.26 MB

TABLE VII: Number of Executed System Calls of Agamotto

Driver	Agamotto	Agamotto + LBR	Deduction
ar5523	7,561	5,624	1,937 (25.6%)
btusb	8,077	5,889	2,188 (27.1%)
go7007	7,731	6,243	1,488 (19.2%)
pn533	7,118	5,774	1,344 (18.9%)
rsi	7,046	5,571	1,475 (20.1%)
si470x	7,452	5,754	1,698 (22.8%)
usx2y	7,226	5,587	1,639 (22.7%)
Average Deduction			22.3%

accuracy of fuzzing inputs.

C. Profiling Accuracy Evaluation

We investigated D-vPMU's efficacy in improving profiling accuracy. We tested its performance in a stressed single VM, multi-VM, and host/VM setting; the latter two have entities that share a CPU, simultaneously profiling using the PMU.

1) Single VM Evaluation: We investigated how virtualization costs contribute to profiling sample loss, which can compromise data reliability and profiling precision. We then evaluated whether D-vPMU's optimizations mitigate this issue and enhance profiling fidelity over KVM.

We profiled the program from Listing 2 using the Linux perf tool, collecting stack traces. Linux's PMI handler writes these traces to a memory-mapped ring buffer, which the user-space perf process then consumes. Linux configures the PMU according to the targeted sampling frequency. A higher frequency generates more PMIs. When the sampling frequency is high and perf cannot consume the buffer in time, the buffer can fill up. This can also result from the perf process being starved for CPU time. If the PMI handler cannot write a new sample due to no buffer space being available, the handler drops the update and reports a sample loss.

In our evaluation, we configured the buffer size to 16 KB and tested with different sampling frequencies, varying between 6,000 to 10,000 Hz. We present the average sample loss over 10 runs in Table VIII. To assess the impact of these losses, we also measured the number of instructions executed during stack trace collection. We present the results with sampling frequency set to 6,000 Hz in Table IX.

Table VIII shows that increasing sampling frequency leads to greater sample loss. KVM suffers from significantly more sample loss than bare metal. We attribute this overhead to PMI delivery latency. As previously mentioned, KVM relies on the host Linux to handle PMIs and then asynchronously injects virtual PMIs into VMs in the PCL. The host functions asynchronously to KVM and might not process a PMI before the VM returns from a hardware PMI exit, causing a delay. This results in a virtual PMI that may not be injected immediately at such VM returns, leading to jitters in delivery latency. Such delays, combined with guest scheduling artifacts, increase the likelihood of sample loss during stack trace updates. In our case, the program being profiled is CPU-intensive, which starves the perf process — perf has a very narrow time window to consume the stack trace from the shared buffer. If the virtual PMI is delivered too late or too early within this window, the PMI handler cannot generate new data, resulting in a sample loss. On the other hand, D-vPMU passes PMIs directly to VMs, eliminating the delays associated with PMI handling and injection. While KVM relies on the host Linux for PMI virtualization, mediated vPMU intercepts PMIs and injects them into VMs, thereby outperforming KVM. However, this method still requires VM exits and introduces greater overhead than direct PMI passthrough.

Table IX presents the number of instructions captured by perf. We used perf's counting mode (no PMIs) to establish a baseline for this expected instruction count value (3B). D-vPMU enables more accurate profiling accuracy than KVM and mediated vPMU and yields similar resulting instruction counts to bare-metal profiling.

2) Multi-VM Evaluation: To evaluate the scalability of D-vPMU's passthrough optimization, we concurrently run two single-core VMs on the same CPU. In each VM, we execute the program in Listing 2 and use Linux perf to profile its execution. Similar to earlier, we evaluated the performance of one VM executing perf with varying sampling frequency. During the evaluation, the other VM executes perf with a 10,000 Hz sampling rate. Since we found that running CPU-intensive workloads concurrently in two VMs degrades perf's performance, we set the ring buffer size of the evaluated VM to 256 KB. Table VIII shows that (in the Multi-VM column), by incorporating PMI passthrough, D-vPMU significantly outperformed KVM and mediated vPMU with much less sample lost count. Table IX shows that D-vPMU sampled more instructions than KVM and mediated vPMU, achieving better profiling fidelity.

3) Host-VM Evaluation: We also quantified the impact of the PMU passthrough mechanism on the Linux host's profiling. We evaluated the performance of two entities, a guest VM and the host, which share a CPU concurrently, utilizing the PMU for profiling. We evaluated the performance of the host executing perf with varying sampling frequency to profile the program in Listing 2. During the evaluation, the VM, employing both of D-vPMU's passthrough mechanisms, executes perf with a 10,000 Hz sampling rate to profile the same program. Like the multi-VM experiment, we configured the perf ring buffer size to 256 KB in the host. The column and row, respectively, under Host-VM in Table VIII and Table IX demonstrate the results of host profiling. As can be

TABLE VIII: Sample Lost Count Comparison

Frequency	Single-VM				Multi-VM			Host-VM [1]		
	KVM	D-vPMU	Mediated	BM	KVM	D-vPMU	Mediated	KVM	D-vPMU	Mediated
10000	35.6	7.8	12.2	2.8	82.9	35.6	53.5	83.4	81.6	80.3
9000	21.4	6.0	10.6	3.0	68.7	34.8	45.2	60.8	62.9	60.8
8000	10.0	4.0	10.4	2.6	42.8	33.2	39.4	59.7	62.3	59.3
7000	9.2	3.2	8.4	2.0	41.7	30.9	37.2	45.4	45.1	45
6000	10.4	3.2	7.8	1.2	34.5	20.5	33	33.9	34.9	33.4

TABLE IX: Sampled Instruction Count Comparison (with 6000 Hz sampling frequency) - Expected 3B

Metric	KVM	D-vPMU	Mediated	BM
Single-VM	2.87 B	2.99 B	2.92 B	3 B
Multi-VM	2.90 B	2.98 B	2.95 B	-
Host-VM[1]	2.95 B	2.96 B	2.96 B	-

seen, incorporating D-vPMU results in a modest performance impact on the host profiling, achieving similar sample losses and profiled instruction count to KVM.

D. Functionality and Security Analysis

In subsection III-A and subsection III-B, we outlined the critical properties that must be considered when implementing MSR and PMI passthrough. To validate our approach, we conducted a series of experiments designed to verify the preservation of these properties.

1) MSR passthrough: We first focus on validating **MSR-P1** (isolating host and VM MSR states) through two basic experiments. In the first experiment, we launched two VMs constrained to run on the same physical CPU. We wrote to an MSR involved in MSR passthrough in the first VM, then wrote to the same MSR in the second VM. Subsequently, we read the MSR value in the first VM. Without the MSR context switching mechanism (see subsection III-A), the first VM could read the value the second VM wrote, compromising VM isolation. Conversely, the first VM correctly read its written value with our MSR context-switching implementation. A similar experiment replacing the second VM with a host process yields identical results, confirming proper MSR isolation and multiplexing between VMs and host processes. These experiments validate that our implementation secures **MSR-P1**, ensuring MSR isolation in virtualized environments.

For **MSR-P2** (Achieving MSR isolation with modest overhead.), we reported the evaluation results of our KVM D-vPMU prototype in earlier sections. We show that implementation incurs modest overhead to VM performance.

2) PMI passthrough: This section discusses our validation that the optimization preserves **PMI-P1** and **PMI-P2**.

PMI-P1 (NMIs are handled by the intended entity). We conducted the following experiments to show that NMIs are handled by the intended entity. First, we addressed performance event skid cases using guest PCL in the VM to generate a fixed number of PMIs. We found that without the solution proposed earlier, which installed a PMI handler in KVM to re-inject misrouted PMIs (see *VM NMIs sent to the host.* in subsection III-B), a PMI could be mistakenly received by the host due to a skid; this leaves the PMI masked in the local APIC, as the host PCL is detached from the vCPU thread. This would prevent the VM from receiving subsequent PMIs. In contrast, our solution effectively mitigates the issue by detecting event skids and injecting the PMI into VMs.

We then evaluated cases where VMs mistakenly handled NMIs intended for the host. We consider both benign and malicious VMs. In this experiment, we used NMIs to dump stack traces, sending them to the processor running the VM and verifying the host's detection capability. For benign VMs, we invoked the hypercall that D-vPMU exposes (see subsection III-B) when encountering an unknown NMI. For malicious VMs, we relied on the *NMI monitor* to check contextual information after VM exit (detailed in subsection III-B). Specifically, to validate safe NMI delivery, we extended the Linux host to check a global bitmap indicating which processors should dump their stack trace and report an NMI if the bit corresponding to the VM's processor is set. In both scenarios, the host detected NMIs for stack trace dumping, proving the effectiveness in ensuring proper NMI handling.

PMI-P2 (isolate NMI/PMI blacking/masking status). We conducted an experiment to validate our NMI unblocking mechanism. We ran a VM and utilized guest PCL to generate a PMI in the VM. We instrumented the guest kernel to force a VM exit via a special hypercall during NMI handling. Within KVM's handler for the hypercall, we used host PCL to generate another PMI. We verified that without running our NMI unblock helper (in Listing 1), NMI blocking remained active during KVM's hypercall handling. The processor cannot receive the second PMI correctly.

We also evaluated the ability of PMI unmasking to ensure that a VM's masking states do not disrupt host processes. We ran a VM and pinned the VM to the same physical CPU. We ran another process, P_{Host}, which executed the program used in subsubsection IV-A3 in host with M = 10,000, N = 100,000,000. We make N larger than earlier to prolong the execution time of P_{Host} (resulting in 10,000 PMIs in our case). After P_{Host} started its execution, we masked PMI in the VM and waited for P_{Host} to complete its execution. Our KVM implementation unmasks PMI upon context switching to P_{Host}. Consequently, PMI is unmasked throughout P_{Host}'s execution. We found that P_{Host} consistently received 10,000 PMIs, successfully preventing prolonged PMI masking.

[1]Performance of the host profiling.

V. Related Work

KVM [31] employs a trap-and-emulate method to virtualize PMU MSR accesses. This approach, however, introduces significant overhead in VM operations. By default, KVM also triggers a VM exit upon receiving a PMI, invoking the host kernel's NMI handler before injecting an NMI into the VM. This PMI handling process further exacerbates the performance overhead. Our work addresses these inefficiencies by simultaneously enabling MSR and PMI passthrough. This dual-pronged approach significantly reduces the overhead associated with KVM's PMU virtualization, enhancing PMU performance within VMs.

Previous work [8] implements MSR passthrough to accelerate PMU virtualization, proposing two timings for MSR context switching: (1) VM entry and VM exit, and (2) vCPU thread switch. The former is achieved via configuring VMCS. The latter switches MSR values only when KVM switches between vCPUs, differing from process context switching (via Linux preempt notifier or host PCL). Notably, [8] does not context switch MSRs between a vCPU and a host process, forbidding the host and VM to share the PMU. In contrast, our MSR passthrough approach multiplexes between VMs and the host with a deferral approach, reducing overhead while preserving the host's profiling capability. Mediated passthrough vPMU [29], [34] also adopts an MSR passthrough approach to optimize KVM's PMU virtualization. Both trap a VM's access to EVTSEL, to secure deployments. However, unlike our work, it does not employ a deferral approach, context switching MSRs on every VM entry and exit, incurring much higher overhead.

None of the prior work [8], [29], [34] supports PMI passthrough. While mediated vPMU [29], [34] reinvents it and installs a dedicated PMI vector to handle PMIs for each VM and perform virtual PMI injection, [8] reuses KVM's PMI virtualization mechanism. These approaches all incur significant overhead from costly VM exits and virtual interrupt injection. In contrast, we employ PMI passthrough to eliminate both and achieve enhanced performance.

ELI [22] enables VMs to handle maskable interrupts without VM exits. While conceptually similar to PMI passthrough, key differences between maskable interrupts and NMIs present unique challenges for PMI passthrough. First, all NMI sources share a single interrupt handler, preventing selective VM exit handling as in [22]'s shadow IDT approach. Second, the lack of NMI processor affinity mechanisms precludes redirecting critical NMIs to dedicated processors. Lastly, NMIs have additional side effects, such as NMI blocking. Directvisor [35] introduced an interrupt passthrough approach, which routes hardware's timer and inter-processor interrupts directly to VMs. However, Directvisor's approach is incompatible with PMI passthrough. It relies on the posted-interrupt mechanism, which does not support NMI deliveries required by PMIs. Further, Directvisor necessitates a one-to-one mapping of physical cores to virtual cores, which severely limits CPU resource scheduling.

VI. Concluding Remarks

Limitations. The use of a deferred MSR context switch prevents host profiling from functioning until the hardware's MSR state is switched back from the VM to the host. This limitation does not affect the profiling capabilities in the host Linux and host applications, although it does prevent host users from profiling the KVM module. Further, Table I shows that the NMI checker imposes overhead to VM exits. We will explore exposing a toggle switch to enable or disable passthrough on demand in the future; this could enhance compatibility and reduce overhead when D-vPMU's passthrough mechanism is unnecessary to VMs. KVM has been recently extended [20] to support the virtualization of Intel's Precise Event Based Sample (PEBS) facility. The current KVM implementation does not leverage PMU or PMI passthrough, potentially causing performance degradation during profiling. We plan to comprehensively evaluate a VM's PEBS usage and optimize its performance in future work.

Conclusion. We introduced D-vPMU, an optimization that incorporates MSR and PMI passthrough to enhance KVM's PMU virtualization performance. These passthrough optimizations enable direct PMU MSR access and PMI handling by VMs, reducing VM exit overhead and complex emulation overhead while retaining KVM's security and functionality. Evaluation of our KVM prototype demonstrates significant improvements in micro-level PMU operations. When a VM profiles popular network applications, Nginx with Linux perf, D-vPMU achieves a significant reduction in KVM's virtualization overhead. In addition, D-vPMU results in much less sample loss than KVM when profiling under stressed and multi-tenant scenarios. To further demonstrate the impact of practical adoption and enable emerging applications, we integrated our optimizations into a KVM-based fuzzing framework to facilitate efficient access to profiling hardware. The optimizations proposed in this work significantly improved the profiling capabilities for practical VM deployments. To benefit the broader community, we plan to open-source the patches for D-vPMU after the paper is accepted.

References

[1] Schneider, F., Payer, M. & Gros, T. Online optimization driven by hardware performance monitoring. *PLDI '07: Proceedings Of The 28th ACM SIGPLAN Conference On Programming Language Design And Implementation*. pp. 373-382 (2007,6)

[2] Yasin, A. A Top-Down Method for Performance Analysis and Counters Architecture. *2014 IEEE International Symposium On Performance Analysis Of Systems And Software (ISPASS)*. pp. 35-44 (2014,3)

[3] Eyerman, S., Eeckhout, L., Karkhanis, T. & Smith, J. A performance counter architecture for computing accurate CPI components. *ASPLOS XII: Proceedings Of The 12th International Conference On Architectural Support For Programming Languages And Operating Systems*. pp. 175-184 (2006,10)

[4] Spisak, M. Hardware-Assisted Rootkits: Abusing Performance Counters on the ARM and x86 Architectures. *WOOT'16: Proceedings Of The 10th USENIX Conference On Offensive Technologies*. pp. 79-90 (2016,8)

[5] Pappas, V., Polychronakis, M. & Keromytis, A. Transparent ROP exploit mitigation using indirect branch tracing. *Proceedings Of The 22nd USENIX Conference On Security*. pp. 447-462 (2013)

[6] Zhou, H., Kang, K. & Yuan, J. HardStack: Prevent Stack Buffer Overflow Attack with LBR. *2019 International Conference On Intelligent Computing, Automation And Systems (ICICAS)*. pp. 888-892 (2019)

[7] Du, J., Sehrawat, N. & Zwaenepoel, W. Performance profiling in a virtualized environment. *Proceedings Of The 2nd USENIX Conference On Hot Topics In Cloud Computing*. pp. 2 (2010)

[8] Du, J., Sehrawat, N. & Zwaenepoel, W. Performance profiling of virtual machines. *Proceedings Of The 7th ACM SIGPLAN/SIGOPS International Conference On Virtual Execution Environments*. pp. 3-14 (2011), https://doi.org/10.1145/1952682.1952686

[9] Menon, A., Santos, J., Turner, Y., Janakiraman, G. & Zwaenepoel, W. Diagnosing performance overheads in the xen virtual machine environment. *Proceedings Of The 1st ACM/USENIX International Conference On Virtual Execution Environments*. pp. 13-23 (2005), https://doi.org/10.1145/1064979.1064984

[10] Nikolaev, R. & Back, G. Perfctr-Xen: a framework for performance counter virtualization. *Proceedings Of The 7th ACM SIGPLAN/SIGOPS International Conference On Virtual Execution Environments*. pp. 15-26 (2011), https://doi.org/10.1145/1952682.1952687

[11] Serebrin, B. & Hecht, D. Virtualizing Performance Counters. *Euro-Par 2011: Parallel Processing Workshops*. pp. 223-233 (2012)

[12] Liu, P., Yang, R., Sun, J. & Liu, X. SysOptic: A Fine-Grained Monitoring System for Virtual Machines Based on PMU. *2019 IEEE International Conference On Service-Oriented System Engineering (SOSE)*. pp. 244-2446 (2019)

[13] Wang, X. & Karri, R. NumChecker: Detecting kernel control-flow modifying rootkits by using Hardware Performance Counters. *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. pp. 1-7 (2013)

[14] Wang, S., Zhang, W., Wang, T., Ye, C. & Huang, T. VMon: Monitoring and Quantifying Virtual Machine Interference via Hardware Performance Counter. *2015 IEEE 39th Annual Computer Software And Applications Conference*. **2** pp. 399-408 (2015)

[15] Liu, W., Liu, X., Li, Z., Liu, B., Yu, R. & Wang, L. Retrofitting LBR Profiling to Enhance Virtual Machine Introspection. *Trans. Info. For. Sec.*. **17** pp. 2311-2323 (2022), https://doi.org/10.1109/TIFS.2022.3183409

[16] Maintainer Perf Wiki. https://perf.wiki.kernel.org/index.php/Main_Page

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

[18] Gregg, B. 2011. https://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html

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

[20] Kleen, A. KVM: x86/pmu: Add support to enable Guest PEBS via DS. (2021), https://lwn.net/Articles/841660/

[21] Kleen, A. Advanced usage of last branch records. (2016), https://lwn.net/Articles/680996/

[22] Amit, N., Gordon, A., Har'El, N., Ben-Yehuda, M., Landau, A., Schuster, A. & Tsafrir, D. Bare-metal performance for virtual machines with exitless interrupts. *Commun. ACM*. 59, 108-116 (2015,12), https://doi.org/10.1145/2845648

[23] Song, D., Hetzelt, F., Kim, J., Kang, B., Seifert, J. & Franz, M. Agamotto: Accelerating Kernel Driver Fuzzing with Lightweight Virtual Machine Checkpoints. *29th USENIX Security Symposium (USENIX Security 20)*. pp. 2541-2557 (2020,8), https://www.usenix.org/conference/usenixsecurity20/presentation/song

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

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

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

[27] Mingwei Zhang and Xiong Zhang, Mediated Passthrough vPMU RFC. (2024), https://github.com/googleprodkernel/linux-kvm/tree/passthrough-pmu-rfc.

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

[29] Zhang, X. KVM: x86/pmu: Introduce passthrough vPM. https://lwn.net/Articles/959653,2024

[30] Xu, L. KVM: x86/pmu: Guest Last Branch Recording Enabling. (2021), https://lwn.net/Articles/844761/

[31] Kivity, A., Kamay, Y., Laor, D., Lublin, U. & Liguori, A. kvm: the Linux virtual machine monitor. *Proceedings Of The Linux Symposium*. **1**, 225-230 (2007)

[32] Corporation, I. PerfMon Events. (2024), https://perfmon-events.intel.com/

[33] Arm Limited, A. CoreSight Architecture. (2024), https://developer.arm.com/Architectures/CoreSight%20Architecture

[34] Zhang, M. Mediated passthrough vPMU for KVM. (2024), https://lpc.events/event/18/contributions/1765/attachments/1502/3311/Mediated%20Passthrough%20vPMU%20for%20KVM.pdf

[35] Cheng, K. and Doddamani, S. and Chiueh, T. and Li, Y. and Gopalan, K.. Directvisor: virtualization for bare-metal cloud. *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. pp. 45–58 (2020), https://doi.org/10.1145/3381052.3381317

[36] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3: System Programming Guide*. Order Number 325384-077US, May 2024.

[37] Dall, C. Li, S.-W, Lim, J.-T, Nieh, J., and Koloventzos, G. "ARM Virtualization: Performance and Architectural Implications," in *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 304–316, ACM/IEEE, 2016. doi:10.1109/ISCA.2016.35

[38] Dall, C. Li, S.-W, Nieh, J. Optimizing the Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC 2017)*, pages 221–234, Santa Clara, CA, July 2017.

[39] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide. Section on Last Branch Record (LBR), 2025. Available at https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.