The Design, Implementation, and Evaluation of Software and Architectural Support for Nested Virtualization on Modern Architectures

Jin Tack Lim

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
under the Executive Committee
of the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2021

Abstract

The Design, Implementation, and Evaluation of Software and Architectural Support for Nested Virtualization on Modern Architectures

Jin Tack Lim

Nested virtualization, the discipline of running virtual machines inside other virtual machines, is increasingly important because of the need to deploy workloads that are already using virtualization on top of virtualized cloud infrastructures. However, nested virtualization performance on modern computer architectures is far from native execution speed, which remains a key impediment to further adoption. My thesis is that simple changes to hardware, software, and virtual machine configuration that are transparent to nested virtual machines can provide near-native execution speed for real application workloads. This dissertation presents three mechanisms that improve nested virtualization performance.

First, we present NEsted Virtualization Extensions for Arm (NEVE). As Arm servers make inroads in cloud infrastructure deployments, supporting nested virtualization on Arm is a key requirement. The requirement has recently been met with the introduction of nested virtualization support for the Arm architecture. We built the first hypervisor using Arm nested virtualization support and show that, despite similarities between Arm and x86 nested virtualization support, performance on Arm is much worse than on x86. This is due to excessive traps to the hypervisor caused by differences in non-nested virtualization support. To address this problem, we introduce a novel paravirtualization technique to rapidly prototype architectural changes for virtualization and evaluate their performance impact using existing hardware. Using this technique, we introduce NEVE, a set of simple

architectural changes to Arm that can be used by software to coalesce and defer traps by logging the results of hypervisor instructions until the results are actually needed by the hypervisor. We show that NEVE allows hypervisors running real application workloads to provide an order of magnitude improvement in performance over current Arm nested virtualization support and up to three times less overhead than x86 nested virtualization. NEVE is included in the Armv8.4 architecture.

Second, we introduce virtual-passthrough, a new approach for providing virtual I/O devices for nested virtualization without the intervention of multiple levels of hypervisors. Virtual-passthrough preserves I/O interposition while addressing the performance problem of I/O intensive workloads as they perform many times worse with nested virtualization than without virtualization. With virtual-passthrough, virtual devices provided by a host hypervisor, the hypervisor that runs directly on the hardware, can be assigned to nested virtual machines directly without delivering data and control through multiple layers of hypervisors. The approach leverages the existing direct device assignment mechanism and implementation, so it only requires virtual machine configuration changes. Virtual-passthrough is platform-agnostic and easily supports important virtualization features such as migration. We have applied virtual-passthrough in the Linux KVM hypervisor for both x86 and Arm hardware, and show that it can provide more than an order of magnitude improvement in performance over current KVM virtual device support on real application workloads.

Third, we introduce Direct Virtual Hardware (DVH), a new approach that enables a host hypervisor to directly provide virtual hardware to nested virtual machines without the intervention of multiple levels of hypervisors. DVH is a generalization of virtual-passthrough

and does not limit virtual hardware to I/O devices. Beyond virtual-passthrough, we introduce three additional DVH mechanisms: virtual timers, virtual inter-processor interrupts, and virtual idle. DVH provides virtual hardware for these mechanisms that mimics the underlying hardware and, in some cases, adds new enhancements that leverage the flexibility of software without the need for matching physical hardware support. We have implemented DVH in KVM. Our experimental results show that combining the four DVH mechanisms can provide even greater performance than virtual-passthrough alone and provide near-native execution speeds on real application workloads.

Table of Contents

List of Figures

List of Tables

Acknowledgements

The entire journey of this Ph.D. would not have been possible without the support of my family, friends, and colleagues. First and foremost, I want to thank my advisor, Jason Nieh. His amazing ability to capture the key ideas and present them in such an interesting and coherent way inspired me every time. Second, I want to thank the rest of my dissertation defense committee: Junfeng Yang, Larry Rudolph, Edouard Bugnion, and Ronghui Gu.

Christoffer Dall has been much more than a colleague; he has been a mentor and has become a sincere friend. He provided me with strong moral support throughout my Ph.D. life. Especially in my hardest times, he encouraged and influenced me to have confidence in my own work and push forward. Christoffer also nurtured me on many technical aspects, particularly while working on the NEVE project. The entire experience helped me level up my skills and abilities. To this day, many great moments from our time working together come to mind. Shih-Wei Li joined Columbia in the same year as me, and we went through some challenging moments together. His keen attention to detail and sharp comments amazed me time and time again. Furthermore, Naser AlDuaij and Alexander Van't Hof generously shared their valuable advice from prior experiences as senior Ph.D. students, which helped me a lot to navigate the Ph.D. program.

Much of my work was done interacting with the Linux and QEMU open-source communities and CloudLab. Marc Zyngier provided helpful code reviews of KVM/ARM patches and took over the KVM/ARM nested virtualization patch series that I had worked on. Peter Xu provided insight to troubleshoot many issues related to passthrough and virtual IOMMU for x86. Eric Auger provided an early version of Arm SMMU emulation support, which allowed me to evaluate virtual-passthrough on Arm. Robert Ricci, Mark Harber, David M. Johnson, and Leigh Stoller helped with many questions related to using the CloudLab infrastructure, where most of my experiments were conducted.

Finally, I'd like to thank my family, who provided outstanding support throughout my journey despite the fact that it started in a later stage of life. My wife, Se Jin, made a huge sacrifice to give up a guaranteed future career in Korea to come to the US to support my dream. She has endured this long period with endless patience as my supporter and as a mother, all while continuing her fight to establish a new professional life. My parents and parents-in-law stepped up to provide the support we needed undergoing countless trips back and forth between Korea and the US. I can't even imagine how I would be where I am today without the amazing support of my family.

Chapter 1

Introduction

Virtualization is a key technology in cloud computing environments. Virtualization enables software that is designed to run directly on hardware, such as operating systems (OSes), to run inside virtual machines (VMs). A VM is an abstraction of the underlying physical machine, and a hypervisor is software that runs on hardware realizing the abstraction, as shown in Figure 1.1(a).

Nested virtualization involves running multiple levels of hypervisors to support running VMs inside VMs, as shown in Figure 1.1(b). We refer to the *host hypervisor* as the first hypervisor that runs directly on the hardware, the *guest hypervisor* as the hypervisor running inside a VM, and the *nested VM* as the VM created by the guest hypervisor. For more levels of virtualization, we refer to the host hypervisor as the L0 hypervisor, the VM created by the L0 hypervisor as the L1 VM, the guest hypervisor as the L1 hypervisor, the hypervisor running on top of the L1 hypervisor as the L2 hypervisor, and so on.

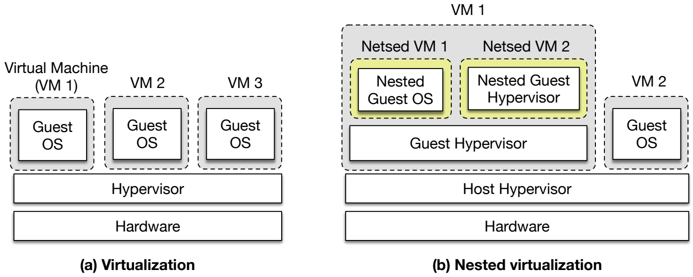


Figure 1.1: Virtualization and Nested Virtualization

Nested virtualization is increasingly important for cloud computing as deploying VMs on top of Infrastructure-as-a-Service (IaaS) cloud providers is becoming more commonplace and requires nested virtualization support [43, 45, 89, 27]. Furthermore, OSes including Linux and Windows have built-in hypervisors to support legacy applications [83] and enhance security [82]; these OS features require nested virtualization support to run in VMs.

While nested virtualization has many benefits, nested virtualization performance on modern computer architectures is far from native execution speed, and this remains a key impediment to its further adoption. In general, the main reason for virtualization overhead is hypervisor interventions during VM execution required to preserve the VM abstraction, which makes software running inside a VM run slower than it would on the physical machine. Current architectural and software support for nested virtualization incur significantly more hypervisor interventions than non-nested virtualization support because multiple levels of hypervisors involved, which leads to the poor nested virtualization performance.

While a VM can execute most instructions without hypervisor interventions, some instructions require hypervisor interventions to provide a proper execution environment for the VM and to protect other software running on the physical machine, such as the hypervisor and other VMs. For example, an instruction to shut down a machine from a VM should not shut down the physical machine, which would affect the hypervisor and other VMs' executions. The hypervisor needs to trap such instructions so that the VM exits its own execution and execution switches to the hypervisor so it can emulate the instruction that caused the trap.

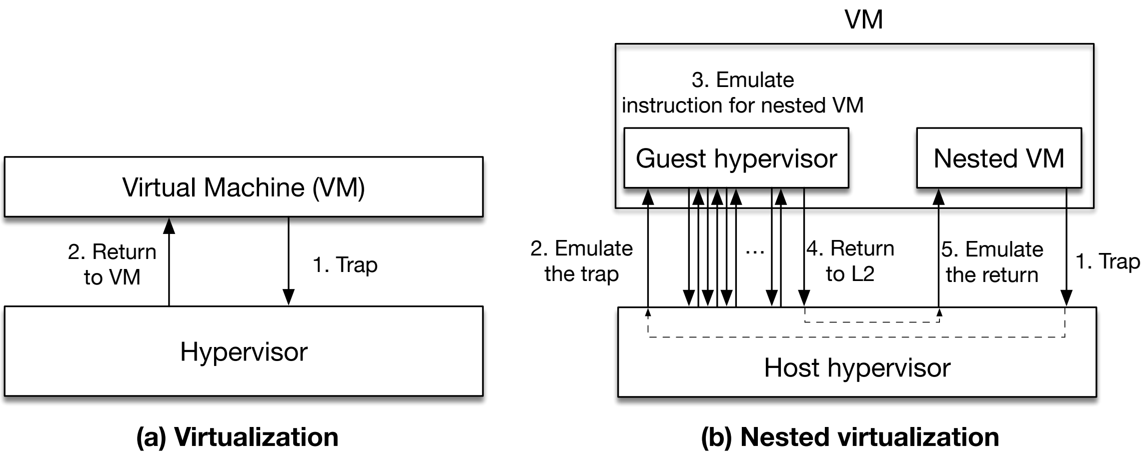


Figure 1.2: Steps to Handle an Exit from a VM and a Nested VM

Modern computer architectures, such as x86 and Arm, provide hardware virtualization support for this well-known trap-and-emulate virtualization technique [22]. On these architectures, a designated hypervisor mode of operation exists where a hypervisor can access architectural features for virtualization. For example, software running in hypervisor mode can execute a special instruction to run a VM. A VM runs in a different mode of operation, non-hypervisor mode. These architectures enable the hypervisor running in hypervisor mode to configure which instructions under what specific conditions are required to trap, allowing the hypervisor to emulate the instruction that VMs attempted to execute. For the previous shutdown example, the shutdown instruction would cause a trap from the VM to the hypervisor, which would then determine the reason for the VM exit was the shutdown instruction and transparently emulate that instruction by terminating only the VM. Figure 1.2(a) shows the steps of handling a trap from a VM.

Modern architectures also use trap-and-emulate to support nested virtualization. Since architectures, such as x86 and Arm, only provide single hypervisor mode, a VM that runs a hypervisor in it still has to stay in non-hypervisor mode. The architectures trap instructions that attempt to access virtualization support inside the VM to hypervisor mode, and the host hypervisor emulates the instructions. As a result, the guest hypervisor can indirectly leverage architectural support for virtualization transparently inside the VM, and can manage its own VMs, nested VMs. The host hypervisor multiplexes the hardware between the guest hypervisor and nested VMs, running both of them inside a VM.

Architectural support for virtualization based on trap-and-emulate incurs only modest performance overhead for non-nested virtualization because the time for emulation, which is based on the number of exits and the cost of handling exits, is small compared to VM execution. The same approach, however, is not efficient enough for nested virtualization. For example, switching between the hypervisor and a VM is a basic and frequent virtualization operation involving more hypervisor interventions for nested virtualization as shown in step 1, 2, 4, and 5 of Figure 1.2(b). The guest hypervisor's attempt to run its own VM, the nested VM, will not switch execution to the nested VM directly but will be trapped to hypervisor mode first because the guest hypervisor does not have permission to access virtualization features required to run a VM. The host hypervisor then switches to the nested VM as part of emulating the guest hypervisor's instruction. Similarly, on exiting the nested VM, execution always first switches to hypervisor mode, and the host hypervisor forwards the exit to the guest hypervisor as part of emulating an exit from the nested VM. Therefore, a switch between a nested VM and the guest hypervisor is indirect and involves twice as many hypervisor interventions compared to non-nested virtualization.

Furthermore, handling a nested VM exit in a guest hypervisor is more costly than handling an exit in a hypervisor for non-nested virtualization because virtualization features that guest hypervisors use will only be provided via trap-and-emulate, as shown in Figure 1.2(b) step 3. For example, accessing VM's states and inspecting its exit reason are frequent hypervisor operations, but they all need to be emulated if they are done by the guest hypervisor. Therefore, each exit from a nested VM can result in many more exits due to indirect hypervisor-VM switches and further virtualization operation emulations, a problem known as exit multiplication, which causes a dramatic increase in virtualization overhead.

My thesis is that simple changes to hardware, software, and virtual machine configuration that are transparent to nested virtual machines can provide near-native execution speed for real application workloads. This dissertation presents three mechanisms that improve nested virtualization performance by reducing the degree of exit multiplication in one of two ways. First, we eliminate the need for the guest hypervisor to exit when executing certain instructions to use virtualization features, avoiding the need to trap and emulate frequent hypervisor operations. This reduces exit multiplication by reducing the number of exits due to guest hypervisor execution. Second, we eliminate the need for the guest hypervisor to handle certain exits from a nested VM, avoiding the need for the host hypervisor to forward such exits to the guest hypervisor. This reduces exit multiplication by avoiding guest hypervisor execution, so it will not cause further exits.

First, we present NEsted Virtualization Extensions for Arm (NEVE) [75]. As Arm servers make inroads in cloud infrastructure deployments, supporting nested virtualization on Arm is a key requirement. The requirement has been met with the introduction of nested virtualization support in the Armv8.3 architecture. We built the first hypervisor using Armv8.3 nested virtualization support and show that, despite similarities between Arm and x86 nested virtualization support, performance on Arm is much worse than on x86.

This is due to excessive traps from the guest hypervisor to the host hypervisor caused by differences in non-nested virtualization support. To address this problem, we introduce a novel paravirtualization technique to rapidly prototype architectural changes for virtualization and evaluate their performance impact using existing hardware. Using this technique, we introduce NEVE, a set of simple architectural changes to Arm, that can be used by software to coalesce and defer traps by logging the results of hypervisor instructions executed by the guest hypervisor in the VM until the results are actually needed by the host hypervisor. This reduces exit multiplication by batching the handling of multiple hypervisor instructions on one exit instead of exiting for each individual hypervisor instruction executed by the guest hypervisor. We show that NEVE allows hypervisors running real application workloads to provide an order of magnitude improvement in performance over the Armv8.3 nested virtualization support and up to three times less overhead than x86 nested virtualization. NEVE is included in the Armv8.4 architecture.

Second, we introduce virtual-passthrough, a new approach for providing virtual I/O devices for nested virtualization without the intervention of multiple levels of hypervisors. Virtual-passthrough preserves I/O interposition while addressing the performance problem of I/O intensive workloads as they perform many times worse with nested virtualization than without virtualization. With virtual-passthrough, virtual devices provided by a host hypervisor can be assigned to nested VMs directly without delivering data and control through multiple layers of hypervisors. Therefore, virtual-passthrough reduces exit multiplication by eliminating the need for guest hypervisor execution when the nested VM interacts with the assigned virtual I/O devices. The approach leverages the existing direct device assignment mechanism and implementation, so it only requires virtual machine configuration changes. Virtual-passthrough is platform-agnostic and easily supports important virtualization features such as migration. We have applied virtual-passthrough in the Linux KVM hypervisor for both x86 and Arm hardware, and show that it can provide more than an order of magnitude improvement in performance over current KVM virtual device support on real application workloads.

Third, we introduce Direct Virtual Hardware (DVH), a new approach that enables a host hypervisor to directly provide virtual hardware to nested VMs without the intervention of multiple levels of hypervisors [73]. DVH is a generalization of virtual-passthrough and does not limit virtual hardware to I/O devices. Beyond virtual-passthrough, we introduce three additional DVH mechanisms: virtual timers, virtual inter-processor interrupts, and virtual idle. DVH provides virtual hardware for these mechanisms that mimics the underlying hardware and, in some cases, adds new enhancements that leverage the flexibility of software without the need for matching physical hardware support. Like virtual-passthrough, DVH reduces exit multiplication by eliminating the need for guest hypervisor execution when the nested VM accesses the virtual hardware. We have implemented DVH in KVM. Our experimental results show that combining the four DVH mechanisms can provide even greater performance than virtual-passthrough alone and provide near-native execution speeds on real application workloads.

Contributions

The contributions of this dissertation include:

1. We build the first hypervisor to use Arm nested virtualization support. We show that

despite similarities between Arm and x86 nested virtualization support, performance on Arm is much worse than on x86.

2. We identify that the performance bottleneck of Arm nested virtualization is due to excessive traps from the guest hypervisor to the host hypervisor, which are caused by the architecture design.
3. We propose NEsted Virtualization Extensions for Arm (NEVE), a set of simple architectural changes to Arm that reduce the number of traps to the host hypervisor significantly.
4. We introduce a novel paravirtualization technique to rapidly prototype architectural changes for virtualization and evaluate their performance impact using existing hardware.
5. We implement and evaluate NEVE using the paravirtualization technique. NEVE provides an order of magnitude better performance than the existing Armv8.3 nested virtualization support.
6. We discuss the inclusion of NEVE in the Armv8.4 architecture as Arm's Enhanced Support for Nested Virtualization.
7. We introduce virtual-passthrough, a new approach for providing virtual I/O devices for nested virtualization without the intervention of multiple levels of hypervisors. Virtual-passthrough preserves I/O interposition while addressing the performance problem of I/O intensive workloads.
8. We have applied virtual-passthrough in the Linux KVM hypervisor for both x86 and Arm hardware, and show that it can provide more than an order of magnitude improvement in performance over current KVM virtual device support on real appli-

cation workloads.

9. We introduce a new PCI device capability, the migration capability, and we implement the capability in a virtio virtual PCI device to support migration of unmodified nested VM using virtual-passthrough.
10. We introduce Direct Virtual Hardware (DVH), a new approach that enables a host hypervisor to directly provide virtual hardware to nested virtual machines without the intervention of multiple levels of hypervisors. DVH is a generalization of virtual-passthrough and does not limit virtual hardware to I/O devices.
11. Beyond virtual-passthrough, we introduce three additional DVH mechanisms: virtual timers, virtual inter-processor interrupts, and virtual idle. We implement them in KVM.
12. We show that combining the four DVH mechanisms can provide even greater performance than virtual-passthrough alone and provide near-native execution speeds on real application workloads while supporting important virtualization features such as migration.
13. We contribute to open source communities by providing source code for the first Arm nested virtualization support in KVM We also provide various bug fixes and reports related to x86 nested virtualization, PCI passthrough, virtual IOMMU, and virtio in KVM/QEMU. The changes are either upstreamed or in the process of being upstreamed.

Chapter 2

NEVE: Nested Virtualization Extensions for Arm

While the x86 architecture has dominated the server and cloud infrastructure markets, the Arm architecture is leveraging its dominance in the mobile and embedded space to make inroads in cloud infrastructure deployments [18]. Because of the demand for nested virtualization in these markets, Arm introduced architectural support for nested virtualization in the Armv8.3 architecture [21]. However, no Armv8.3 hardware supporting nested virtualization exists yet and, as a consequence, no hypervisors have been developed for Arm that support nested virtualization. While nested virtualization can deliver reasonable performance on x86 [19], it remains an unexplored technology on Arm. Given the growing popularity of virtualization on Arm and attractive use cases for nesting, investigating the future for nesting support on Arm is important.

Because of the absence of Arm hardware with nested virtualization support, we introduce a novel approach for evaluating the performance of new architectural features for virtualization using paravirtualization. Paravirtualization is traditionally used to simplify hypervisor design and improve hypervisor performance by avoiding the use of certain architectural features that are difficult or expensive to virtualize. We instead use paravirtualization to enable a hypervisor to leverage new architectural features that do not exist in the underlying hardware by using existing instructions in the underlying architecture to mimic the behavior and performance of new architectural features. The approach enables us to evaluate the performance of new architectural features for virtualization on existing hardware with real application workloads and hypervisors at native execution speeds.

Using this approach, we build the first Arm hypervisor to support nested virtualization. We modified KVM/ARM [34] to support Armv8.3 nested virtualization features. Both the hypervisor design and Armv8.3 are based on a trap-and-emulate approach similar to how software supports nested virtualization on x86 where both architectures have single-level hardware virtualization support. Despite these similarities, we show that Armv8.3 nested virtualization performance is quite poor and significantly worse than x86. Our results provide the first quantitative comparison between Arm and x86 nested virtualization performance, and provide crucial insight regarding virtualization support on other emerging architectures. We identify for the first time how differences in the design of single-level hardware virtualization support, which do not cause significant performance impact for non-nested virtualization, end up causing a very significant performance impact for nested virtualization due to the Arm's RISC-style architecture.

To address this problem, we propose NEsted Virtualization Extensions for Arm (NEVE), a new architecture feature for Arm that can improve nested virtualization performance with minimal hardware and software implementation complexity. We observe that a primary source of overhead for nested virtualization on Arm is the cost of context switching between a VM and the hypervisor and between different VMs. On Arm, there are many instructions involved in these context switches that require hypervisor intervention. This cost is exacerbated when multiple levels of hypervisors are involved in running a VM for nested virtualization. Our insight is that many of these hypervisor instructions do not have an immediate impact on VM or hypervisor execution, but simply prepare the hardware for running a different execution context at a later time. NEVE takes advantage of this insight by logging the results of these hypervisor instructions executed in the VM and coalescing and deferring traps to the hypervisor that runs directly on the hardware until the execution context being affected is actually used, thereby significantly reducing the overhead of nested virtualization. NEVE supports completely unmodified guest hypervisor and OS software.

Using our paravirtualization approach for architecture performance evaluation, we have built a complete hypervisor for nested virtualization by modifying KVM/ARM to use NEVE on existing hardware. Our measurements on real application workloads show that NEVE can provide up to an order of magnitude better performance than the Armv8.3 architecture, and up to three times less overhead than x86 nested virtualization. Arm has revised its nested virtualization architectural support to include NEVE starting with the Armv8.4 architecture [49].

2.1 Architectural Support for Arm Nested Virtualization

The Armv8 architecture [10] includes the Arm Virtualization Extensions (VE). VE adds a more privileged CPU mode, known as an exception level, called EL2. Arm CPU exception levels EL0, EL1, and EL2 are designed to run user applications, an OS kernel, and a hypervisor, respectively. Each exception level has different sets of system registers, which are only accessible from the same or more privileged exception level. For single-level trap-and-emulate virtualization, VMs are executed in EL0 and EL1. CPU virtualization works

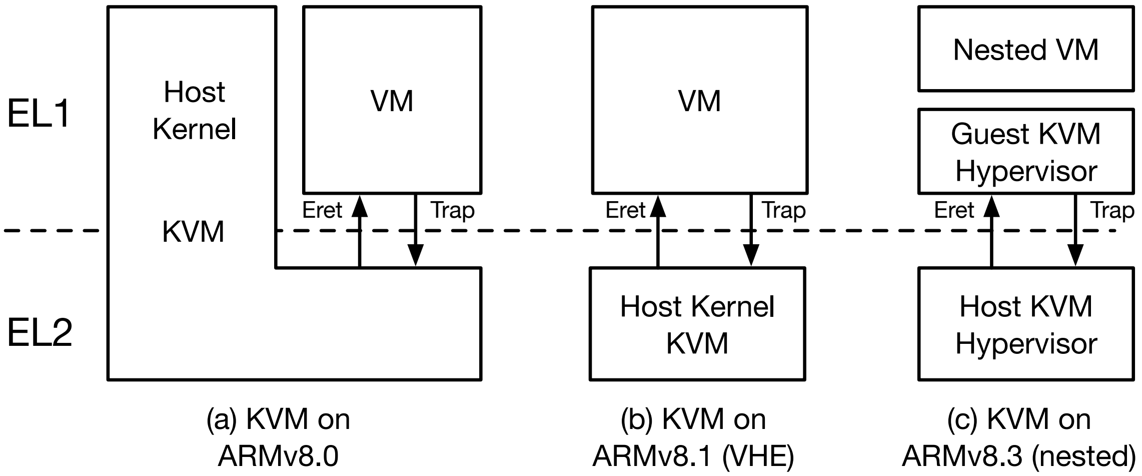


Figure 2.1: Arm Hardware Virtualization Extensions

by letting software executing in EL2 configure the CPU to trap to EL2 on events and instructions that cannot be safely executed by a VM, for example on hardware interrupts and I/O instructions. Memory virtualization works by allowing software in EL2 to point to a set of page tables, Stage-2 page tables, used to translate the VM's view of physical addresses to machine addresses, while Stage-1 page tables can be used and managed by the VM without trapping to the hypervisor to translate virtual to physical addresses. Interrupt virtualization works by allowing the hypervisor to inject virtual interrupts to VMs, which VMs can acknowledge and complete without trapping to the hypervisor.

Armv8.1 introduced the Virtualization Host Extensions (VHE) [30]. Without VHE, hosted hypervisors, which are integrated with an OS kernel, needed to split their OS and hypervisor functionality across EL1 and EL2, respectively [34], as shown in Figure 2.1(a). VHE allows running both the OS kernel and hypervisor functionality in EL2 as shown in Figure 2.1(b). VHE expands the capabilities of EL2 so that it can be functionally equivalent to EL1, including adding additional EL2 system registers. VHE supports running existing OS kernels written for EL1 in EL2 without having to modify the OS source code.

VHE transparently redirects EL1 system register access instructions to access EL2 system registers instead. New instructions are added for the hypervisor to access the EL1 system registers, which belong to the VM context.

Running nested hypervisors on Arm involves running the host hypervisor using EL2 as normal, but *deprivileging* the guest hypervisor to preserve protection so that instead of running in EL2, as it is designed to do, it runs in either EL0 or EL1. While it is functionally possible to run the guest hypervisor in EL0 and trap and emulate hypervisor instructions to the host hypervisor, there are at least two important drawbacks of that approach. First, delivering interrupts to the guest hypervisor has to be fully emulated in software and cannot leverage the VE support for virtual interrupts because the architecture does not support delivering virtual interrupts to EL0. Second, because the host hypervisor must trap hypervisor instructions, it must enable a feature to *Trap General Exceptions* (TGE), which has the unfortunate side effect of disabling the Stage-1 virtual address translations for the guest hypervisor. The host hypervisor must instead construct shadow page tables using Stage-2 translation for the guest hypervisor running in EL0, making the host hypervisor overly complicated and likely results in poor performance.

A better alternative is running the guest hypervisor in EL1. Unfortunately, this does not work without Armv8.3 nested virtualization support. Hypervisor instructions do not trap to EL2 when executed in EL1, but cause exceptions directly to the guest hypervisor in EL1. This would typically lead to an unmodified hypervisor crashing if executed in EL1. For example, suppose the guest hypervisor wishes to configure its own page table base register. Since this EL2 register is accessed using a hypervisor instruction which does not trap to EL2 but instead causes an exception in EL1, attempts to change the register would cause an unexpected exception to the guest hypervisor executing in EL1, likely leading to a software crash. To address this limitation, Arm introduced architectural support for nested virtualization in the Armv8.3 architecture [21]. It works in three parts. First, it enables trapping of hypervisor instructions executed in EL1 to EL2. Second, it disguises the deprivileged execution by telling the guest hypervisor that it runs in EL2 if it reads the CurrentEL register, which contains the current exception level. Third, it supports using the EL2 page table format in EL1. The resulting configuration using KVM on Armv8.3 is shown in Figure 2.1(c).

Comparison to x86 Armv8.3 nested virtualization support is similar to x86 in that guest hypervisor instructions can be configured to trap to the host hypervisor. However, the core hardware virtualization support is different. We limit our discussion of x86 to Intel VT as it is similar to AMD-V for all purposes discussed here. While Arm VE provides a separate CPU privilege level, EL2, with its own set of features and register state, Intel VT provides root vs. non-root mode, completely orthogonal to the CPU privilege levels, each of which supports the same full range of user and kernel mode functionality. Both Arm and Intel trap into their respective EL2 and root modes, but transitions between root and non-root mode on Intel are implemented with a VM Control Structure (VMCS) residing in normal memory, to and from which hardware state is automatically saved and restored when switching to and from root mode, for example when the hardware traps from a VM to the hypervisor. Arm instead, as a RISC-style architecture, has a simpler hardware mechanism to transition between EL1 and EL2, but leaves it up to software to decide what state needs to be saved and restored, providing more flexibility to optimize what is done for each transition.

Because of these differences in the core hardware virtualization support, Armv8.3 must provide some additional mechanisms not necessary for x86 to provide the same level of support for nested virtualization. First, since Arm augments the existing CPU privilege level for virtualization support, as opposed to introducing an orthogonal mechanism, Armv8.3 needs to disguise the CPU privilege level so that a hypervisor that normally runs in EL2 does not know that it is running in EL1 as a guest hypervisor. Second, because EL2 is a separate privilege level with its own page table format that differs from the EL1 page table format, Armv8.3 allows a hypervisor, which would normally use the EL2 page table format when running in EL2, to use the same format when running as a guest hypervisor in EL1.

2.2 Paravirtualization for Architecture Evaluation

Unfortunately, Armv8.3 hardware supporting nested virtualization is not available. As architectural support for virtualization is increasingly common, understanding the performance of these features is important, ideally before they become set in production hardware. However, evaluating new architecture features for virtualization is challenging because of costs associated with prototyping new hardware and the need to understand the interaction of both hardware and software. Chip vendors use cycle-accurate simulators to measure performance, but they are typically many orders of magnitude slower than real hardware, making it hard to evaluate real-life workloads. Booting a full virtualization stack including the hypervisor and VM can take days, and even then, measuring key application performance characteristics such as fast I/O performance using 10G Ethernet is still not possible. Furthermore, simulators of commercial architecture designs are themselves quite complex to build and often closed and proprietary, limiting their availability in practice. Software developers often can only use simpler architecture models before hardware is available, at the cost of not being able to measure any real architecture performance.

To overcome this challenge, we introduce an existing idea, paravirtualization, in a new context. Paravirtualization allows for a software interface to a VM that differs slightly from the underlying hardware [108]. It is used to make hypervisors simpler and faster by avoiding certain architecture features that are complex or difficult to virtualize efficiently. We instead use paravirtualization to allow us to build hypervisors using new architecture features that do not exist on current hardware, and measure the performance of a full virtualization stack using new architecture features at native execution speeds on existing hardware.

Paravirtualization to evaluate new architecture features is only possible when the performance and functionality of the proposed feature can be closely emulated using instructions supported by available hardware. For core virtualization support in the architecture, changes often involve traps; either by adding features to trap on instructions that previously did not trap, or by adding logic to avoid costly traps. In both cases, paravirtualization can be used to replace instructions inside the VM with other ones supported by available hardware such that the resulting behavior and performance closely mimic that of a proposed architectural change.

For example, as discussed in Section 2.1, current Arm server hardware does not support nested virtualization, because when a hypervisor runs inside a VM on top of another hypervisor, various instructions that it executes do not trap to the underlying hypervisor for proper execution, but instead simply fail improperly. However, if we replace those hypervisor instructions with instructions that do trap on current hardware and the trap cost is expected to remain similar in future hardware, we can obtain similar relative performance to future hardware that supports nested virtualization with correct trapping behavior.

There are a couple key assumptions in this example. First, the approach is useful for evaluating the relative performance of an architecture feature compared to something else, not to estimate absolute performance of future hardware. For example, the approach can provide an accurate evaluation of the overhead of nested virtualization compared to native execution.

Second, the approach assumes that certain types of traps are interchangeable in terms of performance. For example, on Arm, the trap cost using an explicit trap instruction should be similar to the cost of any system register access instruction that traps. Only the cost of the trap itself needs to remain similar; the overall cost of handling the respective trap can be quite different. This assumption is likely to be true in most cases and we have validated it on Arm hardware, as discussed in Section 2.4.

Using this approach, it becomes possible to efficiently evaluate the performance of full virtualization stacks interacting with fast I/O peripherals, using many CPU cores, and with real-world workloads. It avoids the extremely slow performance, complexity, and limited availability of cycle-accurate simulators for recent architecture versions of commercial CPUs. Perhaps more importantly, the approach allows co-design and rapid prototyping of software and architecture together, reducing long feedback loops common today when the performance of full software stacks is not known until full OS support and hardware is released, which is long after the architecture design phase takes place.

2.3 KVM/ARM Nested Virtualization for Armv8.3

Because Armv8.3 hardware is not yet available, we leverage our paravirtualization approach discussed in Section 2.2 to allow us to design, implement, and evaluate the first Arm hypervisor to support nested virtualization using Armv8.3 architectural support on existing Armv8.0 hardware, which was the newest publicly available Arm hardware at the time this research was conducted. Since both Arm and x86 provide a single level of architectural virtualization support, we take an approach similar to Turtles [19] for supporting nested virtualization on x86, where multiple levels of virtualization are multiplexed onto the single level of architectural support available. We have implemented nested virtualization support on Arm by modifying KVM/ARM [34], the widely-used mainline Linux Arm hypervisor. There are two kinds of modifications: (1) changes to KVM/ARM as a host hypervisor to support running guest hypervisors on Armv8.3, and (2) paravirtualization of KVM/ARM to run as a guest hypervisor on Armv8.0 with similar behavior as an unmodified KVM/ARM guest hypervisor on Armv8.3. We have posted the former to the Linux KVM community [74], and the patches are being upstreamed by the KVM/ARM maintainer [127].

CPU virtualization is accomplished by deprivileging a guest hypervisor so that instead of running in EL2, it runs in EL1 and traps on hypervisor instructions to the host hypervisor running in EL2, which emulates the instruction as needed. A guest hypervisor and its nested VMs all run in a single VM from the point of view of the host hypervisor. The host hypervisor emulates virtual CPUs, including the virtualization extensions, by providing a virtual EL2 mode, creating the illusion to the guest hypervisor running in the VM, that it runs on real hardware capable of running additional VMs. Once the host hypervisor emulates the full architecture including VE to a VM, nesting is recursively supported. Based on the support from the L0 host hypervisor, the L1 guest hypervisor can provide the same architecture environment in the L2 nested VM to run an L2 hypervisor. An L2 hypervisor will run in EL1 and trap on hypervisor instructions to the L0 host hypervisor, which can then forward it to the L1 guest hypervisor providing the emulated architecture for the L2 hypervisor. In this manner, nested virtualization can be done recursively as each hypervisor is limited to providing the architecture environment including VE for the next level hypervisor running in a VM, but is not concerned with further levels of hypervisors.

To mimic Armv8.3 behavior using Armv8.0 hardware so that hypervisor instructions run by the guest hypervisor trap as needed to the host hypervisor, we paravirtualize the guest hypervisor by replacing the hypervisor instructions with `hvc` instructions. An `hvc` instruction takes a 16-bit operand and generates an exception to EL2, which can read the 16-bit operand back from a system register. We encode the hypervisor instructions using the 16-bit operand so that on the trap to EL2, the host hypervisor is informed of the original guest hypervisor instruction that was replaced by an `hvc` and can emulate the behavior of that instruction.

Our paravirtualization technique can be implemented in multiple ways. We added wrappers around all candidate instructions at the source code level, which, depending on a configuration option, at compile time replaces hypervisor instructions with `hvc` instructions. In this way, we did not change any of the logic or instruction flow of the original KVM/ARM code base and thereby avoided unintentionally introducing bugs or departing from the original hypervisor implementation. It is also possible to paravirtualize the guest hypervisor using a fully automated approach, for example by binary patching a guest hypervisor image.

There are four kinds of hypervisor instructions that are paravirtualized to mimic Armv8.3 behavior so they trap if executed by the guest hypervisor on Armv8.0 hardware. First, instructions that can only run in EL2, such as those that directly access EL2 registers, are undefined when executed in EL1 on Armv8.0, so they are paravirtualized to trap to EL2 to access virtual EL2 state.

Second, instructions that run as part of the hypervisor and access EL1 registers are paravirtualized to trap to EL2 because they will now interfere with the execution of the guest hypervisor which is really running in EL1. For example, an Arm hypervisor will configure EL1 registers to run a VM with its guest OS in EL1. This works fine if the hypervisor is running in EL2 and writes to EL1 registers for the VM, but is problematic if the hypervisor is deprivileged running in EL1, because it will then unknowingly be overwriting its own EL1 register state. Instead, these EL1 access instructions must trap to the host hypervisor which will then emulate the instruction on virtual EL1 register state. The host hypervisor is then responsible for multiplexing EL1 state between the guest hypervisor and the nested VM by context switching the hardware EL1 state to the nested VM's virtual EL1 state when the nested VM runs. For some EL1 access instructions, existing Armv8.0 mechanisms are used by the host hypervisor to configure them to trap, avoiding paravirtualization of these instructions.

Third, the `eret` instruction is paravirtualized to trap to EL2 and reading the `CurrentEL` special register is paravirtualized to return EL2 as the current exception level. `eret` is used by a hypervisor to return to a VM. The guest hypervisor should not directly return to a nested VM without the host hypervisor's intervention, but must trap to the host hypervisor. The nested VM's EL1 register state is emulated by the host hypervisor; entering the nested VM is only possible once the host hypervisor loads the emulated nested VM state to physical registers.

Finally, VHE adds a number of new instructions that are undefined on Armv8.0 which must be paravirtualized to trap to EL2 so they can be emulated. These new instructions are used to access EL1 state when running in EL2 with register access redirection enabled, as explained in Section 2.1. Because these instructions are not defined on Armv8.0, they generate an exception to EL1 when executed by a guest hypervisor, instead of trapping to EL2. To allow guest hypervisors to be configured with VHE on Armv8.0, these instructions are paravirtualized to trap as they would on Armv8.3. Because VHE is designed to make EL2 work the same way as EL1 and because the guest hypervisor already runs in EL1, running a VHE guest hypervisor works trivially without further changes.

Memory virtualization is done using shadow page tables [1] to handle additional levels of memory translation imposed by nested virtualization. Arm hardware supports only two stages of address translation via Stage-1 and Stage-2 page tables. Nested virtualization requires at least three: L2 VM virtual address (VA) to L2 VM physical address (PA), L2 VM PA to L1 VM PA, L1 VM PA to L0 PA. Similar to previous work [19], the host hypervisor creates shadow Stage-2 page tables to map from L2 VM PAs to L0 PAs by collapsing Stage-2 page tables from the guest and host hypervisors. The Stage-1 MMU translates L2 VAs to L2 PAs using the L2 guest OS's page tables, and the Stage-2 MMU then translates L2 VM PAs to L0 PAs using the shadow page tables.

Interrupt virtualization is accomplished by providing a hypervisor control interface to a guest hypervisor via trap-and-emulate. This interface is used by a hypervisor to control virtual interrupts for higher-level VMs and is multiplexed onto the single-level Arm virtual interrupt support in the Arm Generic Interrupt Controller (GIC). When a guest hypervisor programs registers in the hypervisor control interface, this must trap to the host hypervisor to sanitize and translate the payload before writing shadow copies of the register payload into the hardware control interface. The hypervisor control interface is memory mapped with GICv2 and therefore trivially traps to EL2 when not mapped in the Stage-2 page tables, but GICv3 uses system registers and must use paravirtualization of the guest hypervisor to mimic Armv8.3's behavior of trapping EL1 accesses to EL2 on Armv8.0.

2.4 Evaluation of Armv8.3 Nested Virtualization

We present some experimental results that quantify the nested virtualization performance of Armv8.3 based on running our paravirtualized KVM/ARM guest hypervisor on our KVM/ARM host hypervisor on multicore Arm hardware. We also measure the performance of a KVM x86 guest hypervisor on top of a KVM x86 host hypervisor to compare against a more mature nested virtualization solution with a similar hypervisor design; KVM x86 is based on Turtles. These results provide the first measurements of Arm nested virtualization as well as the first comparison of nested virtualization between Arm and x86. Experiments were conducted using server hardware in CloudLab [41].

Arm measurements were done using HP Moonshot m400 servers, each with a 64-bit Armv8-A 2.4 GHz Applied Micro Atlas SoC with 8 physical CPU cores. Each m400 node had 64 GB of RAM, a 120 GB SATA3 SSD for storage, and a Dual-port Mellanox

ConnectX-3 10 GbE NIC. x86 measurements were done using Cisco UCS SFF 220 M4 servers, each with two Intel E5-2630 v3 8-core 2.4 GHz CPUs. Hyperthreading was disabled on the nodes to provide a similar hardware configuration to the Arm servers. Each node has 128 GB of ECC memory (8x16 GB DDR4 1866 MHz dual-rank RDIMMs), a 2x1.2 TB 10K RPM 6G SAS SFF HDD for storage, and a Dual-port Cisco UCS VIC1227 VIC MLOM 10 GbE NIC. The x86 hardware includes VMCS Shadowing [55], the latest x86 hardware support for nested virtualization. All servers were connected via 10 GbE, and the interconnecting network switch easily handles multiple sets of nodes communicating with full 10 Gb bandwidth.

To provide comparable measurements, we kept the software environments across all hardware platforms and hypervisors the same as much as possible. For the host and guest hypervisors, we used KVM in Linux 4.10.0-rc3 with QEMU 2.3.50, with our modifications for Arm nested virtualization. KVM/ARM can be configured to run with or without VHE support; we ran experiments with both versions as guest hypervisors. KVM was configured with its standard VHOST virtio network, and with `cache=none` for virtual block storage devices [66, 99, 51]. All hosts and VMs used Ubuntu 14.04 with the same Linux 4.10.0-rc3 kernel and software configuration. All VMs used paravirtualized I/O using virtio-net and virtio-block over PCI.

We ran experiments in two configurations, in a VM (no nesting) and in a nested VM. The VM was configured with 4 cores and 12 GB RAM running on KVM with 8 cores and 16 GB RAM. The nested VM was configured with 4 cores and 12 GB RAM running on a KVM guest hypervisor with 6 cores and 16 GB RAM running on the host KVM hypervisor with 8 cores and 20 GB RAM. The CPU and memory configurations were

	Armv8.3			**x86**	
Micro-benchmark	**VM**	**Nested VM**	**Nested VM VHE**	**VM**	**Nested VM**
Hypercall	2,729	422,720	307,363	1,188	36,345
Device I/O	3,534	436,924	312,148	2,307	39,108
Virtual IPI	8,364	611,686	494,765	2,751	45,360
Virtual EOI	71	71	71	316	316

Table 2.1: Microbenchmark Cycle Counts

selected to provide the same hardware resources to the VM or nested VM used for running the experiments while ensuring more than adequate hardware resources for the underlying hypervisor(s).

We leveraged the kvm-unit-test microbenchmarks [65] to quantify important micro-level interactions between the hypervisor and its VM. Table 2.1 shows the results for running kvm-unit-test in the VM and nested VM configurations for Armv8.3, with and without VHE, and x86. Measurements are shown in cycles instead of time to provide a useful comparison across hardware. Despite using a similar hypervisor architecture on Arm and x86, which both leverage trap-and-emulate hardware support for nested virtualization, as well as sharing the same architecture-independent parts of the KVM implementation, the measurements show that Armv8.3 has drastically worse nested virtualization performance than x86.

The Hypercall benchmark measures the cost of switching from a VM to the hypervisor, and immediately back to the VM without doing any work in the hypervisor. Compared to using a VM, making hypercalls from a nested VM to a guest hypervisor on Armv8.3 is 155 and 113 times more expensive using a non-VHE and VHE guest hypervisor, respectively. When a nested VM makes a hypercall, it first traps to the host hypervisor running in EL2. The host hypervisor then forwards this hypercall to the guest hypervisor by emulating an exception to the virtual EL2 mode in the VM. When the guest hypervisor processes the hypercall, it simply returns back to the nested VM. However, the process of transitioning between the guest hypervisor and the nested VM involves executing many hypervisor instructions that trap to the host hypervisor, which ends up being very expensive.

The Device I/O benchmark measures the cost of accessing an emulated device in the hypervisor. This is a frequent operation for many device drivers and provides a common baseline for accessing I/O devices emulated in the hypervisor. Device I/O is more costly than Hypercall because it emulates the device in addition to performing similar operations to Hypercall. This additional work reduces the relative overhead of running in a nested VM versus a VM, but the overhead is still hundreds of thousands of cycles on Armv8.3 compared to tens of thousands of cycles on x86.

The Virtual IPI (Inter Processor Interrupt) benchmark measures the cost of issuing a virtual IPI from one virtual CPU to another virtual CPU when both virtual CPUs are actively running on separate physical CPUs. This is a frequent operation in multicore OSes that affects many multithreaded workloads. Virtual IPI is more costly than Hypercall because it involves exits from both the sending VM and receiving VM. The sending VM exits because sending an IPI traps and is emulated by the underlying hypervisor. The receiving VM exits because it gets an interrupt which is handled by the underlying hypervisor. Compared to VMs, virtual IPIs between CPUs in nested VMs are more than 73 and 59 times more expensive using non-VHE and VHE guest hypervisors, respectively.

The Virtual EOI benchmark measures the cost of completing a virtual interrupt, also known as End-Of-Interrupt. The interrupt controllers of both platforms, GIC [9] on Arm and APICv [56] on x86, include support for completing interrupts directly in the VM without trapping to the hypervisor. As a result, this operation is much less expensive than the other benchmarks which trap. The KVM host hypervisor provides support on both Arm and x86 so that nested VMs can use hardware-accelerated virtual interrupt completion, resulting in the same cost for both VMs and nested VMs.

In all cases except Virtual EOI, the cost of running the microbenchmarks in a nested VM on Armv8.3 is prohibitively expensive compared to running in a VM. Compared to x86, nested VM performance on Armv8.3 imposes more than an order of magnitude more overhead in terms of cycle counts, and up to 7 times more overhead in terms of relative performance compared to a VM. While trap-and-emulate nested virtualization provides reasonable performance on x86, it does not on Armv8.3.

To investigate the reasons behind the poor Armv8.3 performance, we measured the average number of traps to the host hypervisor when running the Hypercall benchmark. While Hypercall only causes a single trap when running in a VM, it causes 126 and 82 traps to the host hypervisor when running in a nested VM using a non-VHE and VHE guest hypervisor, respectively. Clearly, each trap, also known as an exit, from the nested VM results in a multitude of additional traps from the guest hypervisor to the host hypervisor. This is a major source of overhead for nested virtualization and is called the exit multiplication problem [19].

The guest hypervisor using VHE performs better than without VHE, because it traps less often. When KVM/ARM runs with VHE enabled, it uses EL1 system register access instructions wherever possible with the expectation that the hardware redirects these instructions to EL2 registers, as discussed in Section 2.1. When this is done as a VHE guest hypervisor running in EL1 on Armv8.0 hardware, it simply accesses EL1 registers directly without trapping to the host hypervisor, and the host hypervisor configures the EL1 hardware registers with the guest hypervisor's state. In contrast, a non-VHE guest hypervisor can only access EL2 state using EL2 system register access instructions, and each such access will trap to the host hypervisor since EL2 registers are not accessible at EL1. Despite this reduction in the number of traps for a VHE guest hypervisor, its nested virtualization performance remains poor.

Our measurements of Armv8.3 nested virtualization performance are based on replacing guest hypervisor instructions on Armv8.0 that do not trap as they would on Armv8.3 with `hvc` instructions, which are explicit trap instructions, to mimic Armv8.3 behavior. The replaced instructions are mostly system register access instructions along with a few `eret` instructions. On Arm, the cost of a trap should be evaluated in two parts: (1) finding out that you need to generate an exception, and (2) generating the exception. The first can range from expensive (memory fault) to being free (`hvc` instruction), with a system register trap being almost free. The second is a fixed cost for all instructions. As a result, the cost of traps for the replaced instructions is expected to be very similar to that of an `hvc` instruction on all implementations of the Arm architecture.

We further measured the trap cost of several different system register access instructions that trap on Armv8.0 hardware and compared their cost with an `hvc` instruction. In all cases, trapping from EL1 to EL2 was between 68 to 76 cycles, and returning from a trap to EL2 back to EL1 was 65 cycles. The difference in trap costs across different instructions was less than 10% overall and less than 10 cycles. These measurements on Armv8.0 hardware support our assumption that `hvc` instructions can be used as a suitable replacement to mimic Armv8.3 instructions that trap on system register accesses with similar performance.

2.5 NEVE: NEsted Virtualization Extensions

Nested virtualization support as introduced in Armv8.3 traps hypervisor instructions from a deprivileged guest hypervisor running in EL1 to a host hypervisor running in EL2. A single exit from a nested VM can result in the guest hypervisor issuing many hypervisor instructions, resulting in a multitude of additional traps from the guest hypervisor to the host hypervisor. Many hypervisor instructions need to trap because they access system registers. If we can reduce the number of accesses to system registers that need to trap, we can potentially reduce overhead and improve the performance of nested virtualization on Arm.

System registers accessed by the guest hypervisor can be loosely classified into two groups: VM registers, which only affect the VM, and hypervisor control registers, which directly affect hypervisor execution. A key observation is that VM registers do not have an immediate effect on the guest hypervisor's execution, but instead are used to prepare the hardware for running the nested VM when execution returns to the nested VM.

Based on this observation, we propose NEVE, an addition to the Armv8.3 architecture that avoids traps from the guest hypervisor to the host hypervisor for a wide range of hypervisor instructions that access system registers. NEVE supports unmodified guest hypervisors, both hosted and standalone designs, and unmodified guest OSes. NEVE has three key mechanisms. First, it avoids traps to the host hypervisor for VM registers and instead adds hardware support to store VM registers in memory until they are actually needed for VM execution. In Armv8.3, when a guest hypervisor accesses a VM system register, it traps to the host hypervisor, which simply stores this value in memory in a software-managed data structure, and later programs this value into physical registers when running the nested VM. NEVE instead supports this operation in hardware by using an architecturally defined storage format and transparently rewriting system register access instructions into normal memory accesses.

Second, NEVE reduces traps to the host hypervisor for hypervisor control registers by instead identifying and using equivalent registers that can be accessed without trapping. In Armv8.3, when the guest hypervisor writes to a hypervisor control register and traps to the host hypervisor, in many cases, the host hypervisor handles the trap by writing into an equivalent EL1 register. For example, the guest hypervisor will write the base address of the exception vector for itself in VBAR_EL2 which will trap to the host hypervisor, which in turn needs to write the address to VBAR_EL1, the equivalent EL1 register, so that the guest hypervisor running in EL1 will handle exceptions correctly. In cases where the EL1 and EL2 registers have the same format, NEVE instead supports this operation in hardware by transparently redirecting accesses to EL2 registers to EL1 registers without trapping to the host hypervisor.

Third, NEVE reduces traps to the host hypervisor when reading certain hypervisor control registers by keeping a cached copy in memory and redirecting register read instructions into normal memory accesses. Read instructions, in the absence of side effects, have no immediate impact on hypervisor execution and can be serviced from a memory cache to avoid traps.

2.5.1 Architecture Specification

NEVE introduces an EL2 Virtual Nested Control Register (VNCR_EL2) which is managed exclusively by the host hypervisor. The host hypervisor can use the VNCR_EL2 to enable and disable NEVE and to configure a *deferred access page* in memory used to store the values of VM system registers. Table 2.2 shows the bit fields in the VNCR_EL2 register. The BADDR field contains the physical base address of the deferred access page. The layout of the deferred access page can be arbitrarily defined as long as each VM system register is stored at a well-defined offset from BADDR. The Enable bit completely enables or disables NEVE. When the Enable field is set to 1, and the Armv8.3 nested virtualization support is enabled, all accesses to the VM system registers which would otherwise trap to the host hypervisor are redirected to memory accesses to the deferred access page. Similarly, the register redirection described above for hypervisor control registers is enabled and disabled using the Enable field in the VNCR_EL2.

Fields	**Description**
bits[52:12]	BADDR: Deferred Access Page Base Address
bits[11:1]	Reserved
bit[0]	Enable

Table 2.2: VNCR_EL2 Register Fields

It is up to the host hypervisor to determine when NEVE is enabled and when register values are copied to and from the deferred access page. In a typical workflow, the host hypervisor populates the deferred access page with the initial values of the registers and enables NEVE before running the guest hypervisor. During guest hypervisor execution, all accesses to VM system registers are redirected to the deferred access page. When the host hypervisor needs to use the VM register values, it simply accesses the deferred access

Category	Register	Description
VM Trap Control	HACR_EL2	Hypervisor Auxiliary Control
	HCR_EL2	Hypervisor Configuration
	HPFAR_EL2	Hypervisor IPA Fault Address
	HSTR_EL2	Hypervisor System Trap
	TPIDR_EL2	EL2 Software Thread ID
	VMPIDR_EL2	Virtualization Multiprocessor ID
	VNCR_EL2	Virtual Nested Control
	VPIDR_EL2	Virtualization Processor ID
	VTCR_EL2	Virtualization Translation Control
	VTTBR_EL2	Virtualization Translation Table Base
VM Execution Control	AFSR0_EL1	Auxiliary Fault Status 0
	AFSR1_EL1	Auxiliary Fault Status 1
	AMAIR_EL1	Auxiliary Memory Attribute Indirection
	CONTEXTIDR_EL1	Context ID
	CPACR_EL1	Architectural Feature Access Control
	ELR_EL1	Exception Link
	ESR_EL1	Exception Syndrome
	FAR_EL1	Fault Address
	MAIR_EL1	Memory Attribute Indirection
	SCTLR_L1	System Control
	SP_EL1	Stack Pointer
	SPSR_EL1	Saved Program Status
	TCR_EL1	Translation Control
	TTBR0_EL1	Translation Table Base 0
	TTBR1_EL1	Translation Table Base 1
	VBAR_EL1	Vector Base Address
Thread ID	TPIDR_EL2	Software Thread ID

Table 2.3: VM System Registers

page. For example, when the guest hypervisor runs the nested VM, it executes the `eret` instruction to enter the nested VM, which traps to the host hypervisor. The host hypervisor copies register values from the deferred access page to physical EL1 registers to run the nested VM and disables NEVE while running the nested VM so the VM can access its EL1 registers. Similarly, when the host hypervisor emulates an exception from the nested VM to the guest hypervisor, it copies the EL1 system register values from the hardware into the deferred access page, enables NEVE, and runs the guest hypervisor. The guest hypervisor can now access the VM system registers directly without trapping to the host hypervisor.

NEVE	EL2 Register	Description
Redirect to *_EL1	AFSR0_EL2	Auxiliary Fault Status 0
	AFSR1_EL2	Auxiliary Fault Status 1
	AMAIR_EL2	Auxiliary Memory Attribute Indirection
	ELR_EL2	Exception Link
	ESR_EL2	Exception Syndrome
	FAR_EL2	Fault Address
	SPSR_EL2	Saved Program Status
	MAIR_EL2	Memory Attribute Indirection
	SCTLR_EL2	System Control
	VBAR_EL2	Vector Base Address
Redirect to *_EL1 (VHE)	CONTEXTIDR_EL2	Context ID
	TTBR1_EL2	Translation Table Base 1
Trap on write	CNTHCTL_EL2	Counter-timer Hypervisor Control
	CNTVOFF_EL2	Counter-timer Virtual Offset
	CPTR_EL2	Architectural Feature Trap
	MDCR_EL2	Monitor Debug Configuration
Redirect or trap	TCR_EL2	Translation Control
	TTBR0_EL2	Translation Table Base

Table 2.4: Hypervisor Control Registers

Table 2.3 lists the 27 VM system registers we identified as part of the Armv8.3 specification which do not affect the execution of the hypervisor directly. When enabled, NEVE redirects accesses to these registers to the deferred access page. The VM Trap Control registers control when certain operations performed by the VM trap to the hypervisor and other virtualization features such as Stage-2 translation and virtual interrupts. The VM Execution Control registers are system registers that belong to the VM itself and do not affect hypervisor execution. The Thread ID register, TPIDR_EL2, is commonly used by hypervisors to store thread-specific data but does not affect the hypervisor's execution.

We distinguish two types of hypervisor control registers, normal system registers and GIC registers related to the hypervisor control interface used for interrupt virtualization, discussed in Section 2.3. When the guest hypervisor executes in virtual EL2, which really runs in EL1, accesses to these EL2 registers would normally trap to the host hypervisor, but

NEVE uses two techniques to avoid traps, register redirection and cached copies. Table 2.4 shows the 17 normal system registers we identified that affect the hypervisor's execution in EL2, and the techniques NEVE used to avoid traps.

Register redirection transparently redirects accesses from an EL2 register to its corresponding EL1 register if it exists and has the same format as the EL2 register. Since the guest hypervisor is really running in EL1, EL2 register accesses can be redirected to corresponding EL1 registers such that changes to the registers have the same impact on the hypervisor's execution when running deprivileged in EL1 as running in EL2 on real hardware. NEVE provides register redirection for 12 EL2 registers with corresponding EL1 registers as shown in Table 2.4, two of which are grouped separately (VHE) as they were added as part of VHE and are only relevant for VHE hypervisors.

Cached copies (shown as "Trap on write" in Table 2.4) transparently change reads from EL2 registers that don't have an equivalent EL1 to instead read a cached copy from the deferred access page. The host hypervisor copies the value of the virtual EL2 register to the deferred access page when running the guest hypervisor to cache the latest value of the register for reads from the guest hypervisor. Writes to these registers will trap, allowing the host hypervisor to update the content of the deferred access page as needed. Cached copies are used for four EL2 registers, two of which have similar EL1 registers but with different formats and thus cannot be used with register redirection from EL2 to EL1 registers, namely CNTHCTL_EL2 and CPTR_EL2.

Table 2.4 lists two EL2 registers, TCR_EL2 and TTBR0_EL2, that may be redirected to corresponding EL1 registers for VHE guest hypervisors only. VHE changes the format of these EL2 registers to be identical to the corresponding EL1 registers. VHE guest

NEVE	GIC Register	Description
Trap on write	ICH_HCR_EL2	Hypervisor Control
	ICH_VTR_EL2	VGIC Type
	ICH_VMCR_EL2	Virtual Machine Control
	ICH_MISR_EL2	Maintenance Interrupt Status
	ICH_EISR_EL2	End of Interrupt Status
	ICH_ELRSR_EL2	Empty List Register Status
	ICH_AP0R<n>_EL2	Active Priorities Group 0, n=0-3
	ICH_AP1R<n>_EL2	Active Priorities Group 1, n=0-3
	ICH_LR<n>_EL2	List, n=0-15

Table 2.5: Hypervisor Control GIC Registers

hypervisors can therefore access these registers directly using EL1 access instructions. A non-VHE guest hypervisor, however, would use the EL2 register formats, which are incompatible with the EL1 registers, and therefore the EL2 register accesses cannot be redirected to EL1 registers but must instead be supported using cached copies, trapping on writes to these registers.

Table 2.5 shows the GIC registers in the hypervisor control interface registers we identified that affect the hypervisor's execution in EL2. NEVE uses cached copies in the deferred access page for all of these registers to avoid traps.

Arm also provides performance monitoring, debugging, and timer system registers. We note that accesses to the PMUSERENR_EL0 and PMSELR_EL0 performance monitor control registers can be redirected to the deferred access page like VM system registers, reads from the MDSCR_EL1 debug control register can be redirected to a cached copy so that only writes must trap, and all accesses to the virtual and physical hypervisor timer EL2 registers, namely CNTHV_CTL_EL2, CNTHV_CVAL_EL2, CNTHV_TVAL_EL2, CNTHP_CTL_EL2, CNTHP_CVAL_EL2, and CNTHP_TVAL_EL2, trap as reads must access the registers directly to obtain correct values updated by hardware.

2.5.2 Recursive Virtualization

NEVE supports multiple levels of nesting, also known as recursive nesting. As discussed in Section 2.3, recursive nesting is supported with Armv8.3, because the host hypervisor emulates the same virtual execution environment as the underlying machine including the hardware virtualization support and nesting support. NEVE can further improve the performance of each level of hypervisor. The L0 host hypervisor can create a VM with support for NEVE, which the guest hypervisor will use when running the L2 guest hypervisor. When the L1 guest hypervisor configures NEVE by accessing the VNCR_EL2, we cache the register state to the deferred access page. Because the VNCR_EL2 of the L1 guest hypervisor does not affect the execution of L1 hypervisor, but only affects the execution of the L2 guest hypervisor. On entry to the L2 VM's virtual EL2, the L0 host hypervisor can emulate the behavior of NEVE by using the hardware features directly. This works by translating the VM physical address written by the L1 guest hypervisor into a machine physical address and using this address in the hardware VNCR_EL2. This allows transparently changing register accesses performed by the L2 guest hypervisor into memory and EL1 register accesses. The memory used is provided by the L1 guest hypervisor which can therefore directly access the content of the deferred access page used to support the L2 guest hypervisor running NEVE. In this scenario, NEVE avoids the same amount of traps between the L2 and L1 guest hypervisors as in the normal nested case described above.

2.5.3 Architectural Impact

NEVE represents a relatively small architectural change. It requires adding the VNCR_EL2 register and adding logic to redirect system register access instructions from VM registers to memory at a specified offset when NEVE is enabled in the VNCR_EL2 register. It also requires adding logic to redirect instructions accessing EL2 registers to corresponding EL1 registers or to memory on read accesses, when NEVE is enabled. Since Armv8.1 already supports redirecting system register access instructions to other system registers depending on a run-time configuration, the most invasive part of our proposal is to redirect a system register access to a memory access. To simplify the logic to handle this, we propose that the architecture mandates that the host hypervisor software programs a page-aligned physical address in the VNCR_EL2.BADDR field to avoid the need to perform alignment checks or handle address translation faults.

2.5.4 Implementation

Although NEVE is designed to work with unmodified guest hypervisors, it requires modest hardware changes to do so. To show how NEVE can be used in the absence of a hardware implementation of NEVE, we describe how we can modify KVM/ARM to use this feature via our paravirtualization approach from Section 2.2. We can use the same KVM/ARM design from Section 2.3, but with modifications to CPU virtualization to use NEVE. To implement the deferred access page, we establish a shared memory region between the host and guest hypervisor. We modify KVM/ARM to run as a guest hypervisor using NEVE by replacing instructions that access VM registers with normal load and store instructions that access the shared memory region. We also modify KVM/ARM to run as a guest hypervisor by replacing instructions that access EL2 hypervisor control registers with instructions that access corresponding EL1 registers to provide the equivalent register redirection functionality shown in Table 2.4. The resulting guest hypervisor eliminates the same traps to the host hypervisor and provides the same performance characteristics as a hardware system with NEVE.

We run KVM/ARM in two configurations as the guest hypervisor, non-VHE and VHE. Non-VHE KVM/ARM issues EL1 system register access instructions to access EL1 VM system registers and EL2 system register access instructions to access EL2 VM system registers. These are replaced with load and store instructions to mimic NEVE. As described in Section 2.1, a VHE hypervisor takes advantage of the VHE register redirection feature to allow its integrated OS written for EL1 to run in EL2 without modification. With VHE, EL1 system register access instructions are redirected to EL2 system registers, and KVM/ARM with VHE uses EL1 system register access instructions wherever possible to access EL2 registers, as discussed in Section 2.4. VHE KVM/ARM running as the guest hypervisor will therefore access its own virtual EL2 register state directly using EL1 system register instructions, and there is no need to replace any of these instructions. However, VHE introduces separate EL12 system register access instructions to access EL1 VM system registers, which are replaced with load and store instructions to mimic NEVE.

2.5.5 Performance Impact

The performance benefit of NEVE depends on the design and implementation of the guest hypervisor. The more often a guest hypervisor accesses system registers, the greater potential performance benefit. We briefly discuss three alternative Arm hypervisor designs in this context, which are also the most widely-used Arm hypervisors: KVM/ARM without VHE, KVM/ARM with VHE, and Xen.

First, consider a legacy KVM/ARM implementation without support for VHE [34]. KVM/ARM saves and restores all the VM system registers and modifies VM trap control registers on every VM exit because it uses the same EL1 hardware state to run the Linux kernel portion of the hypervisor. Furthermore, a non-VHE hosted hypervisor frequently accesses the hypervisor control registers when moving between EL1 and EL2. Each of these register accesses from the guest hypervisor traps, resulting in significant exit multiplication using Armv8.3, and NEVE provides a significant performance gain for this hypervisor design as shown in Section 2.6.

Second, consider KVM/ARM in the context of the Virtualization Host Extensions (VHE) [30], which were introduced in Armv8.1. While KVM/ARM was originally designed to run across both EL1 and EL2, VHE allows the KVM/ARM hypervisor to run entirely in EL2. As a result, KVM/ARM no longer needs to use EL1 system registers, and the hypervisor is unaffected by VM trap controls. Therefore, switching between the VM and a VHE hypervisor no longer requires saving and restoring the full VM system register state or configuring VM trap-control registers. However, even with VHE, the current KVM/ARM implementation frequently accesses the VM system registers. The reason is that KVM/ARM saves the VM EL1 context and modifies the VM trap-control registers when switching from the VM to the hypervisor and back, because avoiding these operations while preserving backwards compatibility with non-VHE systems is difficult and would complicate the code base. Furthermore, saving and restoring the full EL1 system register state is still needed when switching between VMs. Therefore, KVM/ARM and similar VHE-enabled hypervisors will benefit from NEVE as shown in Section 2.6.

Third, consider Xen [118] which runs only in EL2 as a standalone hypervisor. Since Xen does not need to use the VM system registers for its execution, it does not save and restore them for every VM exit. However, even Xen must save and restore all the VM system registers when it switches between VMs, which is a common operation on Xen because all I/O is handled in a special separate VM called Dom0. Furthermore, Xen frequently accesses the hypervisor control registers which trap when Xen is a guest hypervisor under Armv8.3. Therefore, Xen is likely to also benefit from NEVE.

2.6 Evaluation of NEVE Nested Virtualization

We measured the nested virtualization performance of NEVE based on running our paravirtualized KVM/ARM guest hypervisor on our KVM/ARM host hypervisor on multicore Arm hardware. An actual hardware implementation of NEVE would not require paravirtualization and would run unmodified guest hypervisors; paravirtualization is only used to provide measurements on Armv8.0 hardware. We also compare NEVE against both Armv8.3 and x86 nested virtualization. Experiments were conducted using the same hardware and software configurations as discussed in Section 2.4. For NEVE measurements, the guest hypervisor has been paravirtualized to use NEVE by sharing a memory region with the host hypervisor for logging the results of hypervisor instructions, and redirecting hypervisor control register accesses to the corresponding EL1 system registers, as discussed in Section 2.5. Although the Arm hardware we used has a GICv2 which uses a memory-mapped interface for registers instead of the GICv3 hypervisor control system registers discussed in Section 2.5, the programming interfaces for both GIC versions are almost identical.

2.6.1 Microbenchmark Results

We repeated the kvm-unit-test microbenchmark measurements from Section 2.4 using NEVE with the same nested VM configurations. Table 2.6 shows the results in terms of cycle counts and relative overhead compared to running in a non-nested VM, along with the previous results from Table 2.1. NEVE provides a dramatic performance improvement compared to Armv8.3. When running in a nested VM, NEVE provides up to 5 times faster performance than Armv8.3 for both non-VHE and VHE guest hypervisors. While x86 nested virtualization remains much faster in terms of absolute cycle counts, this is due to the fact that the base VM measurements are faster on x86 than on Arm. However, comparing the relative performance of a nested vs. non-nested VM on each platform, we see that a guest hypervisor using NEVE has similar overhead to x86. For example for Hypercall, NEVE incurs a 34 to 37 times slowdown while x86 incurs a 31 times slowdown running in a nested vs. non-nested VM.

Table 2.7 shows the average number of traps to the host hypervisor when running each

	Armv8.3		NEVE		x86
Microbenchmark	**Nested VM**	**Nested VM VHE**	**Nested VM**	**Nested VM VHE**	**Nested VM**
Hypercall	422,720 (155x)	307,363 (113x)	92,385 (34x)	100,895 (37x)	36,345 (31x)
Device I/O	436,924 (124x)	312,148 (88x)	96,002 (27x)	105,071 (30x)	39,108 (17x)
Virtual IPI	611,686 (73x)	494,765 (59x)	184,657 (22x)	213,256 (25x)	45,360 (16x)
Virtual EOI	71 (1x)	71 (1x)	71 (1x)	71 (1x)	316 (1x)

Table 2.6: Microbenchmark Cycle Counts

microbenchmark in the nested VM. NEVE reduces the number of traps by more than six times compared to Armv8.3. For example, Hypercall takes only one trap from a VM, but from a nested VM on Armv8.3, it requires 126 and 82 traps to the host hypervisor using a non-VHE and VHE guest hypervisor, respectively. Using NEVE, Hypercall only requires 15 traps to the host hypervisor using either a non-VHE or VHE guest hypervisor. Although non-VHE and VHE guest hypervisors require the same number of traps for Hypercall, they incur different numbers of cycles as shown in Table 2.6 as the traps incurred are different with different emulation costs. For example, VHE adds an additional timer, the EL2 virtual timer, where non-VHE systems only have one virtual timer, the EL1 virtual timer. This additional timer must be supported for VHE guest hypervisors. Because of the register redirection functionality of VHE, and because the VHE guest hypervisor runs deprivileged in EL1, the VHE guest hypervisor directly accesses the EL1 virtual timer when it programs its EL2 virtual timer. However, when attempting to program its EL1 virtual timer, the guest hypervisor will use new VHE-specific EL02 access instructions, which always trap to the host hypervisor, resulting in traps for a VHE guest hypervisor that do not occur for a non-VHE guest hypervisor. As the number of traps also depends on the

	Armv8.3		NEVE		x86
Microbenchmark	**Nested VM**	**Nested VM VHE**	**Nested VM**	**Nested VM VHE**	**Nested VM**
Hypercall	126	82	15	15	5
Device I/O	128	82	15	15	5
Virtual IPI	261	172	37	38	9
Virtual EOI	0	0	0	0	0

Table 2.7: Microbenchmark Average Trap Counts

guest hypervisor implementation especially on Arm, we confirmed that a more optimized VHE guest hypervisor [31] with NEVE reduces the number of traps to the host hypervisor down to 2, which is even less than x86.

The Device I/O and Virtual IPI microbenchmarks show similar improvements. Virtual EOI remains unaffected because the nested VM can interact directly with the hardware support in all cases. The results show how NEVE significantly improves nested virtualization performance by resolving the exit multiplication problem.

2.6.2 Application Benchmark Results

To provide a more realistic measure of performance, we next evaluated nested virtualization using widely-used CPU and I/O intensive application workloads, as listed in Table 2.8. We used three different configurations for our measurements: (1) native: running natively on Linux capped at 4 cores and 12 GB RAM, (2) VM: running in a 4-way SMP guest OS with 12 GB RAM using KVM as a hypervisor with 8 cores and 16 GB RAM, and (3) nested VM: running in a 4-way SMP nested guest OS with 12 GB RAM using KVM as the guest hypervisor, which is capped with 6 cores with 16 GB RAM, while the host KVM hypervisor has 8 cores and 20 GB RAM. The last two configurations are the same as those used in Section 2.4. For benchmarks that involve clients interacting with the server,

Kernbench	Compilation of the Linux 3.17.0 kernel using the allnoconfig for Arm using GCC 4.8.2.
Hackbench	`hackbench` [93] using Unix domain sockets and 100 process groups running with 500 loops.
SPECjvm2008	`SPECjvm2008` [97] 2008 running real life applications and benchmarks chosen to measure Java Runtime Environment performance; we used 15.02 release of the Linaro AArch64 port of OpenJDK.
Netperf	`netperf` v2.6.0 [60] server running with default parameters on the client in three modes: TCP_RR, TCP_STREAM, and TCP_MAERTS, measuring latency and throughput, respectively.
Apache	`Apache` v2.4.7 Web server running `ApacheBench` [100] v2.3 on the remote client, measuring requests handled per second serving the 41 KB file of the GCC 4.4 manual using 10 concurrent requests.
Nginx	`Nginx` v1.4.6 Web server running `Siege` [58] v3.0.5 on the remote client, measuring requests handled per second serving the 41 KB file of the GCC 4.4 manual using 8 concurrent requests.
Memcached	`memcached` v1.4.14 using the `memtier` benchmark v1.2.3 with its default parameters.
MySQL	`MySQL` v14.14 (distrib 5.5.41) running `SysBench` v.0.4.12 using the default configuration with 200 parallel transactions.

Table 2.8: Application Benchmarks

the client ran on a separate dedicated machine and the server ran on the configuration being measured, ensuring that clients were never saturated during any of our experiments. Clients ran natively on Linux with the same kernel version and userspace as the server and configured to use the full hardware available.

Figure 2.2 shows the performance measurements for each VM and nested VM configuration across two different vertical scales given the large dynamic range of the measurements. Since we are most interested in overhead and in comparing across different hardware platforms, VM and nested VM performance are normalized relative to their respective Arm or x86 native execution, with lower meaning less overhead. Table 2.9 shows the non-normalized results from the measurements.

As expected, running in a nested VM on Armv8.3 shows the highest overhead, in some

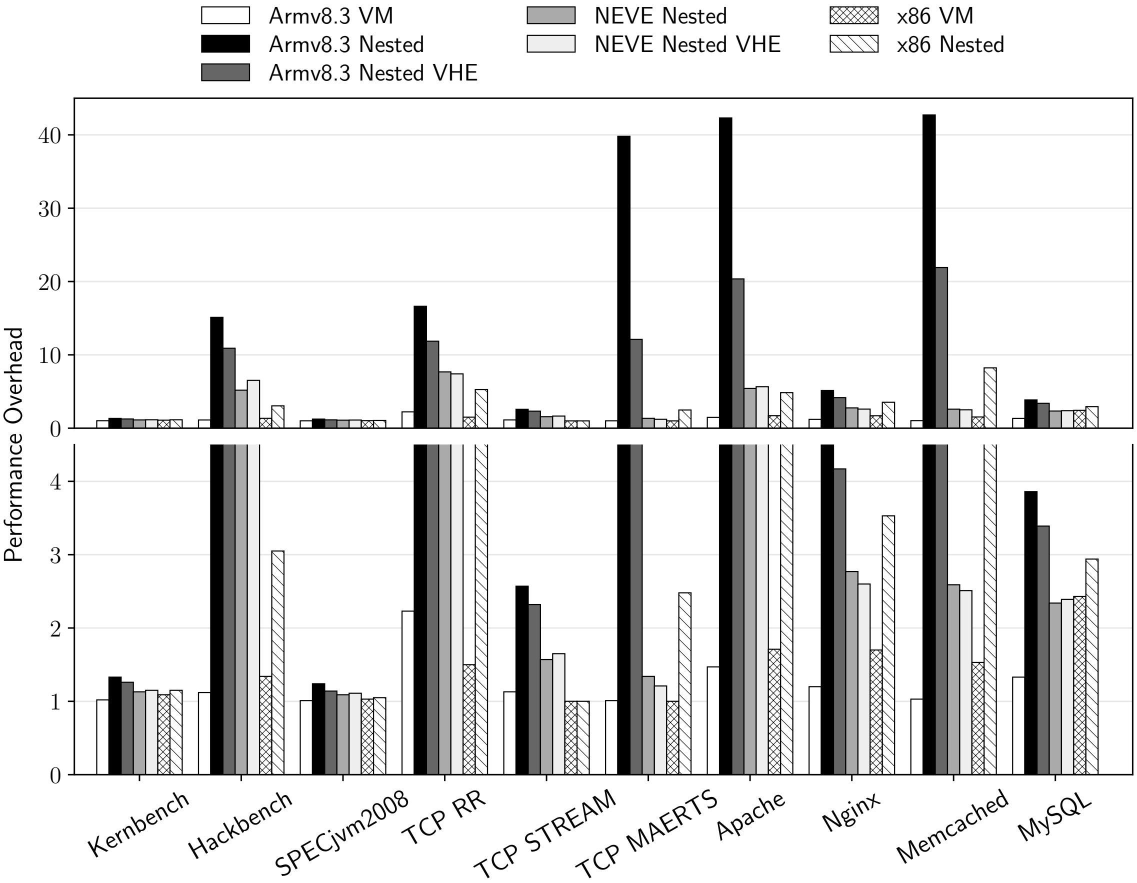


Figure 2.2: Application Benchmark Performance

cases more than 40 times native execution. The largest overhead occurs for network-related workloads, including Netperf TCP_MAERTS, Apache, and Memcached. The high overhead is likely due to the high frequency of interrupts caused by many incoming network packets. Injecting a high number of virtual interrupts to the nested VM results in a high number of switches between the nested VM and the guest hypervisor, which in turn results in many traps using only Armv8.3. Hackbench also performs quite poorly as it is 15 and 11 times slower for non-VHE and VHE guest hypervisors, respectively, compared to native execution. Hackbench is a highly parallel SMP workload in which the OS frequently sends IPIs to synchronize and schedule tasks across CPU cores. As shown in Table 2.6, virtual

Application	Unit	Architecture	Baseline	L1	L2	L2 (VHE)
Kernbench	sec	Armv8.3	49.6	51.1	65.9	62.4
		NEVE	-	-	56.2	57.3
		x86	22.1	24.0	25.40	
Hackbench	sec	Armv8.3	15.8	18.0	239.2	172.5
		NEVE	-	-	82.0	103.3
		x86	6.2	8.3	18.98	
SPECjvm2008	ops/min	Armv8.3	62.2	61.3	50.2	56.8
		NEVE	-	-	54.4	56.1
		x86	163.1	158.6	155.10	
Netperf TCP RR	trans/sec	Armv8.3	23,544	13,533	1,417	1,986
		NEVE	-	-	3,067	3,176
		x86	37,453	25,002	7,103	
Netperf TCP STREAM	Mbits/sec	Armv8.3	9,407	9,385	3,657	4,058
		NEVE	-	-	5,991	5,709
		x86	9,410	9,408	9,394	
Netperf TCP MAERTS	Mbits/sec	Armv8.3	6,242	6,071	236	515
		NEVE	-	-	4,651	5144
		x86	9,414	9,406	3,803	
Apache	trans/sec	Armv8.3	8,255	6,700	195	405
		NEVE	-	-	1,523	1,460
		x86	16,925	9,893	3,489	
Nginx	trans/sec	Armv8.3	2,327	1,947	452	558
		NEVE	-	-	841	897
		x86	6,303	3,698	1,783	
Memcached	trans/sec	Armv8.3	147,754	133,668	3,459	6,742
		NEVE	-	-	57,064	58,850
		x86	463,102	301,821	56,272	
MySQL	sec	Armv8.3	14.8	17.1	57.0	50.0
		NEVE	-	-	34.5	35.3
		x86	7.3	17.6	21.3	

Table 2.9: Application Benchmark Raw Performance

IPIs are costly in nested VMs on Armv8.3, which accounts for the noticeable slowdown in Hackbench. Compared to native execution, CPU-intensive workloads such as SPECjvm and kernbench have a relatively modest performance slowdown in nested VMs, 24% and 33% overhead for a non-VHE guest hypervisor and 14% and 26% for a VHE guest hypervisor, respectively. These workloads have much less overhead than other application workloads because they cause far fewer interactions between the nested VM and the guest hypervisor, and therefore don't suffer as much from the exit multiplication problem as network related benchmarks.

In contrast, NEVE provides significantly better Arm nested virtualization performance, reducing performance overhead by more than or close to an order of magnitude in some cases. For example, Memcached performance goes from more than a 40 times slowdown using Armv8.3 to less than a 3 times slowdown using NEVE, more than an order of magnitude improvement. For network-related workloads including Netperf TCP MAERTS, Apache, and Memcached, NEVE successfully reduces exit multiplication by coalescing traps to reduce the performance overhead. Unlike Armv8.3, which has significantly worse performance, NEVE provides overall performance that is comparable to, and in many cases better than, x86 nested virtualization using the latest x86 virtualization optimizations.

In fact, NEVE incurs significantly less overhead than both Armv8.3 and x86 on many of the network-related workloads, including Netperf TCP MAERTS, Nginx, Memcached, and MySQL. MySQL runs better with NEVE because of the high cost of x86 non-nested virtualization compared to Arm, but this is not the case for the other workloads. For example, Memcached running in a nested VM on x86 shows an 8 times slowdown compared to only a 2.5 times slowdown on NEVE. The reason for this is that Memcached incurs substantially more exits on x86 than Arm, including more than four times as many exits from the nested VM for processing I/O on x86 versus NEVE. Since the relative cost of nested VM exits is similar on x86 and NEVE as shown in Table 2.6, the much higher number of exits on x86 results in much higher overhead than NEVE. Netperf TCP MAERTS and Nginx exhibit similar behavior.

The reason for the much higher number of exits can be explained based on the network

I/O behavior. When a nested VM wants to send packets, its frontend driver notifies the backend driver running in the L1 VM, which causes an exit from the nested VM. Virtio, which is used for paravirtualized I/O, provides mechanisms to optimize I/O performance by reducing the number of VM exits due to notifications. While the backend driver is busy, it tells the frontend driver that it can continue to send packets without further notification. Only once the backend driver has nothing left to do does it tell the frontend driver to notify it again when it has more packets to send. On x86, it turns out that Memcached requires many more virtio notifications than on NEVE. This is because as soon as the backend driver running in L1 is notified, it handles the packets quickly and enables the notification again. In other words, the quicker the backend driver handles packets, the more the frontend driver needs to notify. In fact, by introducing some delay by busy waiting to artificially slow down the backend driver in L1 when running Memcached in x86, we can reduce the x86 Memcached overhead to be close to NEVE. The reason that the x86 L1 backend driver is much faster to process packets than Arm backend driver is that the x86 hardware is much faster than the Arm hardware used. Memcached runs natively roughly three times faster on the x86 server compared to the Arm server. This leads to an interesting performance anomaly that having faster hardware can result in more virtualization overhead.

Our results are based on paravirtualizing KVM/ARM as a guest hypervisor to mimic the behavior of Armv8.3 and NEVE on existing Armv8.0 hardware. Future Arm hardware, such as Armv8.3 hardware, may have somewhat different performance characteristics. In particular, Armv8.3 is a more complex architecture than Armv8.0, so it would not be surprising if the relative cost of traps is higher for such hardware compared to Armv8.0 hardware. Because NEVE improves performance by reducing the number of traps for nested virtualization, a high trap cost for actual Armv8.3 hardware would only accentuate the performance difference between NEVE and Armv8.3, making Armv8.3 nested virtualization performance worse and NEVE's relative improvement even better.

As further validation of this work, we have presented these results to Arm, which has decided to include NEVE in the next release of the Arm architecture.

2.7 Enhanced Support for Nested Virtualiztion

Arm has incorporated NEVE into the Armv8.4 architecture as its Enhanced Support for Nested Virtualization [11]. The Enhanced Support for Nested Virtualization provides two register redirection mechanisms - one to redirect VM register accesses to memory accesses, and the other to redirect EL2 system register accesses to corresponding EL1 system register accesses.

The Enhanced Support for Nested Virtualization adds a single bit to the hypervisor control register (HCR_EL2), the NV2 bit. The Enhanced Support for Nested Virtualization maintains complete backwards compatibility with Armv8.3 nested virtualization; if the NV2 bit is not set, Armv8.4 hardware nested virtualization support behaves the same as Armv8.3. However, setting the NV2 bit significantly changes how the nested virtualization extensions work. In the following, we explain the two most important features of the Enhanced Support for Nested Virtualization.

First, the Enhanced Support for Nested Virtualization introduces a new register, VNCR_EL2, that holds the base memory address used for memory redirection of system register accesses, as we proposed in Section 2.5. A subtle difference between NEVE and the Enhanced Support for Nested Virtualization is that NEVE has the enable bit of the NEVE feature at the bit field 0 of the VNCR_EL2 register to make the register self-contained while the bit in the Enhanced Support for Nested Virtualization locates in the HCR_EL2 register (NV2 bit) where other VM control bits exist.

Second, Enhanced Support for Nested Virtualization redirects system register accesses when the NV2 bit is set to 1. When the guest hypervisor accesses VM registers in EL1, hardware transforms the system register access instructions into memory access instructions. The address of the resulting memory access is defined using a combination of a base address in VNCR_EL2 and a pre-defined offset, which is unique for each register. GIC registers in the hypervisor control interface are also redirected to memory. In addition, when the guest hypervisor accesses EL2 system registers, hardware redirects EL2 register accesses to corresponding EL1 register accesses.

While the Enhanced Support for Nested Virtualization can support unmodified guest hypervisors, the host hypervisor must be modified to use the new hardware features such as the VNCR_EL2 register and the NV2 bit. The host hypervisor needs to allocate memory to keep VM register states and set the memory address to the VNCR_EL2 register so that the hardware can access. When entering the guest hypervisor, the host hypervisor sets the NV2 bit in the HCR_EL2 register to enable register redirections. On the other hand, the host hypervisor clears the NV2 bit when entering a nested VM so that the register accesses from the OS running inside the nested VM work as expected without being redirected to memory.

2.8 Related Work

Paravirtualization is used to make hypervisors simpler and faster by avoiding certain architecture features that are complex or difficult to virtualize efficiently [15, 92]. It is also used to provide virtual architectures that differ from the underlying hardware architecture and can run custom guest OSes designed for performance and scalability [108]. We leverage paravirtualization in a new way to emulate the behavior and measure the performance of new architecture features at native execution speeds on existing and currently available hardware.

Previous work has explored ways to use existing hardware to emulate new hardware. For example, Shade [28] proposed a dynamic translation framework that could run SPARCv9 binaries on a SPARCv8 CPU. However, Shade incurs significant performance overhead. Simulating SPARCv9 on SPARCv8 is more than an order of magnitude slower than native execution on SPARCv8. Our paravirtualization technique is applied statically and does not incur substantial performance overhead, but is focused on virtualization hardware support rather than emulating entire future architectures.

Much work on nested virtualization has focused on x86 [4, 19, 126, 62]. Turtles [19] was the first to show that trap-and-emulate nested virtualization provides reasonable performance on x86. Our Arm hypervisor design uses the same approach as Turtles for CPU and memory virtualization, but uses paravirtualized I/O in lieu of direct device assignment as used in Turtles; the latter was not supported on the Arm server hardware available for our measurements. However, we show that the lessons learned from trap-and-emulate nested virtualization on x86 may not apply to other architectures and that a similar approach on

Arm performs poorly due to differences between the Arm and x86 virtualization support. We introduce a new architecture extension to address this problem and significantly improve Arm performance.

To optimize nested virtualization further, Intel added a new hardware extension called VMCS shadowing [55], which allows a guest hypervisor to execute VMCS access instructions without trapping. VMCS shadowing redirects instructions that are designed to access the VMCS, which is stored in memory, to a different memory location. Our x86 measurements in Section 2.6.2 show that the VMCS shadowing optimization provides roughly a 10% performance improvement. Both VMCS shadowing and NEVE use the basic idea of redirection to mitigate the exit multiplication problem by reducing traps from guest hypervisors. However, unlike VMCS shadowing, NEVE introduces register redirection, and rewrites system register accesses to memory accesses or to other existing registers based on a classification of the functionality of the registers. NEVE is designed for RISC architectures without adopting techniques similar to VMCS which are more suitable for CISC architectures. Unlike the modest gain of VMCS shadowing on x86, NEVE provides an order of magnitude performance improvement on Arm. This is due in part to the CISC vs. RISC architecture designs. x86 automatically saves and restores VM state using the hardware VMCS mechanism which coalesces accesses to VM register state when changing between root and non-root mode in a single operation, mitigating the exit multiplication problem and reduces the benefit of VMCS shadowing. In contrast, Arm requires software to save and restore VM state to individual registers, which results in many more accesses to VM state in software, for which NEVE can significantly reduce exit multiplication and improve performance.

Xen-Blanket [110] leverages nested virtualization to transform existing heterogeneous cloud infrastructures into a homogeneous Blanket layer to host x86 nested VMs. Unlike the aforementioned nested x86 solutions that use hardware virtualization primitives exposed to the guest hypervisor, it does not rely on the host hypervisor to expose those primitives to the nesting layers. Therefore, Xen-Blanket only supports paravirtualized guest OSes, not unmodified OSes in the nested VM.

Agesen et al. [2] proposed software techniques for avoiding VM exits by leveraging existing work on binary translation to detect and rewrite sequences of instructions that cause multiple exits from the VM and rewrite them into translated sequences that only cause a single exit. LeVasseur et al. [69] proposed pre-virtualization, a form of static paravirtualization that uses a hypervisor-specific module in the guest OS to rewrite itself when loaded by a hypervisor. In contrast, NEVE is a hardware approach to transparently rewrite deferrable register accesses to memory accesses in the guest hypervisor and delivers substantial performance gain for workloads running in nested VMs.

Some techniques [3, 37] reduce VM exits by coalescing interrupts, effectively changing the hardware semantics to reduce interrupt overhead while increasing interrupt latency. NEVE does not defer interrupts but defers trapping on instruction execution in a way that preserves existing architecture semantics and improves performance even in the absence of interrupts.

Various Arm virtualization approaches have been developed [48, 54, 32, 35, 17, 34, 118, 104, 8, 31], but none of them support nested virtualization. Our work presents the first Arm hypervisor to support nested virtualization, and introduces new architecture improvements that can be used by host hypervisors to significantly enhance performance.

As virtualization continues to be of importance, understanding the trade-offs of different approaches to hardware virtualization support is instrumental in the design of new architectures. For example, RISC-V [91] is an emerging architecture for which virtualization support is being explored. NEVE provides an important counterpoint to x86 practices and shows how acceptable nested virtualization performance can be achieved on RISC-style architectures.

2.9 Summary

We presented the first in-depth study of Arm nested virtualization. We introduce a novel paravirtualization technique to evaluate the performance of new architectural features before hardware is readily available. Using this technique, we evaluate Armv8.3 nested virtualization support and find that its performance is prohibitively expensive compared to normal virtualization, despite its similarities to x86 nested virtualization. We show how differences between Arm and x86 in non-nested virtualization support end up causing significant exit multiplication on Arm. To address this problem, we introduce NEVE, simple architecture extensions that provide register redirection, and coalesce and defer traps by logging system register accesses from the guest hypervisor to memory and only copying the results of those accesses to hardware system registers when necessary. This reduces exit multiplication by batching the handling of multiple hypervisor instructions on one exit instead of exiting for each individual hypervisor instruction executed by the guest hypervisor. We evaluate the performance of NEVE and show that NEVE can improve nested virtualization performance by an order of magnitude on real application workloads compared to the Armv8.3 architecture, and can provide up to three times less virtualization overhead than x86. NEVE is straightforward to implement in Arm and has been included in the Arm architecture starting with Armv8.4.

Chapter 3

Virtual-passthrough: Boosting I/O Performance for Nested Virtualization

Despite both x86 and Arm architectures having architectural support to enhance nested virtualization performance, many I/O intensive applications still perform many times worse with nested virtualization than they do with non-nested virtualization or native execution without virtualization. A significant portion of the overhead for the applications running inside nested VMs comes from delivering data and control through multiple layers of virtual I/O devices, which involves expensive switches between different virtualization levels.

One solution is device passthrough, also know as direct device assignment. Passthrough directly assigns physical devices to the nested VM so that the nested VM and the physical device can interact with each other without the intervention of multiple layers of hypervisors [19, 23]. For example, the physical device can deliver data directly to the nested VM. However, direct device assignment comes with a significant cost, the loss of I/O interposition and its benefits. I/O interposition allows the hypervisor to encapsulate the state of the VM and decouple it from physical devices, enabling important features such as suspend/resume, live migration [119, 20], I/O device consolidation, and various VM memory optimizations [22]. Many of these features, especially migration, are essential for cloud computing deployments. Furthermore, direct device assignment requires additional hardware support such as physical Input/Output Memory Management Units (IOMMUs) and Single-Root I/O Virtualization (SR-IOV), which may not be available or supported on all platforms, especially in the case of newer virtualization architectures such as Arm. Because of the disadvantages of direct physical device assignment, paravirtual I/O devices are most commonly used in VM deployments. Unfortunately, nested virtualization with virtual I/O devices including paravirtual I/O devices incurs high overhead.

We introduce virtual-passthrough, a novel yet simple technique for boosting I/O performance when using nested virtualization. Virtual-passthrough is similar to direct physical device assignment but instead assigns virtual I/O devices to nested VMs. Virtual devices provided by the host hypervisor can be assigned to nested VMs directly without delivering data and control through multiple layers of virtual I/O devices, which makes I/O operations from nested VMs efficient. Therefore, virtual-passthrough reduces exit multiplication by eliminating the need for guest hypervisor execution when the nested VM interacts with the assigned virtual I/O devices. Virtual-passthrough preserves I/O interposition in the host hypervisor different from physical device passthrough while virtual-passthrough also can easily support important I/O interposition benefits such as migration in the hypervisors at intermediate layers. Scalability is not a problem as many virtual devices can be supported by a single physical device. Supporting both paravirtual and emulated I/O devices is straightforward. The technique is platform agnostic, does not require hardware support such as physical IOMMUs or SR-IOV. Furthermore, virtual-passthrough makes it possible to support virtualization optimizations only in software and add support for new features not natively supported by hardware. For example, virtual-passthrough can provide support for virtual IOMMUs and posted interrupts, even in the absence of corresponding hardware support.

With virtual-passthrough, the host hypervisor makes available its virtual I/O devices for direct assignment to nested VMs. Each intervening hypervisor layer passes through the virtual I/O device to the next layer. The nested VM provides a device driver to communicate with the passed through virtual I/O device, which appears to the nested VM no different from any other I/O device that it accesses. Virtual IOMMUs are made available and used by the nested hypervisors to provide necessary mappings between different guest physical address spaces to support transferring data between the memory buffers of the nested VM and the virtual I/O device provided by the host hypervisor. Data is transferred directly between the nested VM and the virtual I/O device provided by the host hypervisor without further intervention by intermediate layers of virtual machines and hypervisors. Interrupts also can be delivered from the virtual I/O device to the nested VM directly as well as the nested VM can program the virtual I/O device with support from the host hypervisor.

We have implemented virtual-passthrough in KVM for both x86 and Arm hardware and Xen for x86, demonstrating the technique can be used across different architectural platforms and hypervisors. We have evaluated its performance in nested virtualization environments on both x86 and Arm and show that it can provide more than an order of magnitude better performance than current virtual device support on real application workloads. We also show that virtual-passthrough can provide comparable performance to device passthrough while at the same time enabling migration of nested VMs, thereby providing a combination of both good performance and key virtualization features not possible with device passthrough.

3.1 I/O Virtualization for Nested Virtualization

I/O virtualization There are largely two I/O virtualization models, the virtual I/O device model and the passthrough (or direct device assignment) model, as shown in Figures 3.1(a) and 3.1(b), respectively.

In the virtual I/O device model, physical I/O devices are not visible to a VM. Instead, the VM interacts with virtual I/O devices provided by the hypervisor. Each I/O request such as sending a network packet or reading a file is trapped to the hypervisor, which is referred to as I/O interposition. The hypervisor processes the request in software, typically leveraging underlying physical devices, and sending an interrupt to the VM to notify it when the I/O request has been completed. Even though such requests from a VM need to trap to the hypervisor, any memory accesses from the virtual I/O device to access I/O data in the VM can be done asynchronously without trapping as the hypervisor can access VM memory freely. The hypervisor can provide either emulated I/O devices [98], where the VM is not aware that the given device is emulated, or paravirtualized I/O devices [15, 92], where hypervisors and VMs communicate via simplified software I/O interfaces to overcome the inefficiency of I/O device emulation.

The virtual I/O device model is widely used for VMs because it provides tremendous flexibility as a software solution. I/O interposition made possible using the virtual I/O device model brings many benefits [107, 22], including the ability to consolidate many virtual I/O devices on a single physical device as part of server consolidation, thereby increasing utilization, improving efficiency, and reducing costs. It also enables memory optimizations and state encapsulation to facilitate migration, a key virtualization feature.

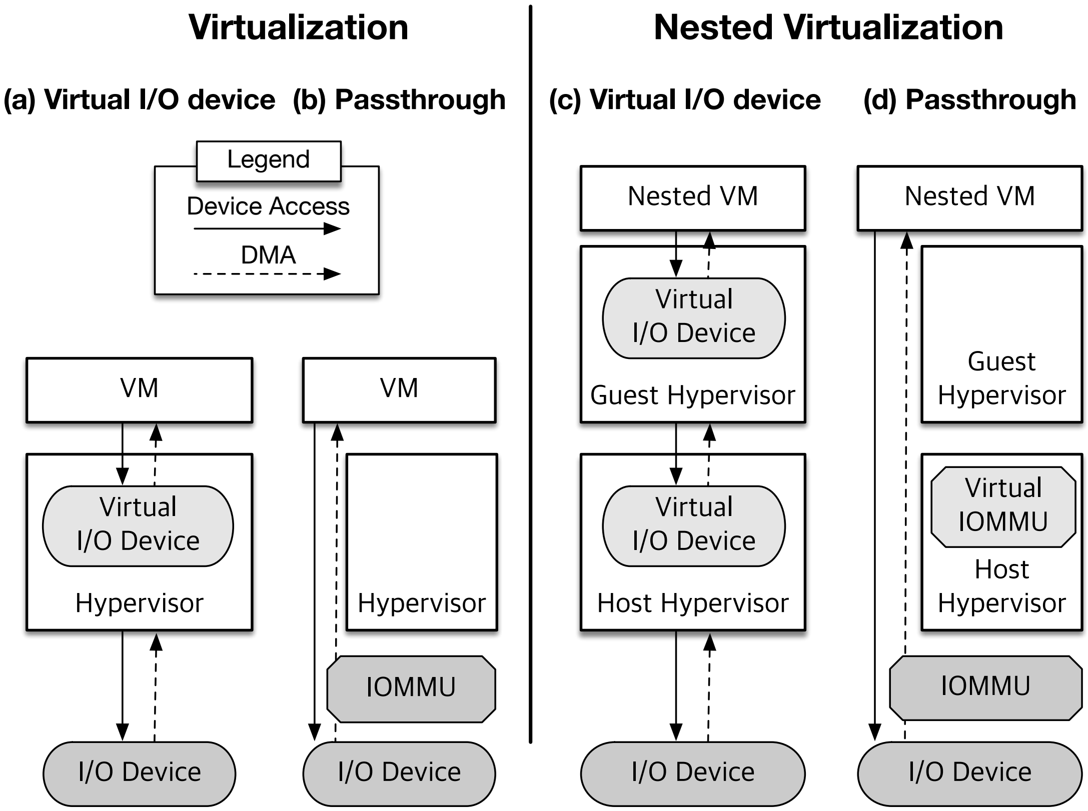


Figure 3.1: I/O Virtualization Models

In the passthrough model, a VM is directly assigned a physical I/O device exclusively, allowing it to access the device without hypervisor intervention, at the expense of I/O interposition. The assigned device can also access VM memory directly as needed to read and write I/O data. Recent hardware support can allow interrupts from the physical I/O device to be delivered to VMs directly without going through the hypervisor. With those direct interactions between the device and the VM, the passthrough model can achieve near-native I/O performance. However, because I/O interposition is lost, it forgoes those benefits, including making it difficult, if not impossible, to support migration. The passthrough model requires additional hardware support. An IOMMU, an address translation unit for I/O devices, is essential for safe and correct direct memory access (DMA) from the assigned physical device to the VM. SR-IOV is needed for scalability, otherwise a device is tied to and can only be used by one VM.

Nested I/O virtualization

For nested I/O virtualization, the hypervisor at each level can decide the I/O virtualization model for its own VM. The most common approach is repeating the model that is provided by the host hypervisor recursively. Using the virtual I/O device model, the hypervisor at each level provides its own virtual devices to its VMs, as shown in Figure 3.1(c). Using the passthrough model, the host hypervisor assigns devices to the guest hypervisor, and the guest hypervisor at each level, in turn, assigns the same devices to its VMs [19], as shown in Figure 3.1(d). These two models naturally inherit and intensify the pros and cons of their non-nested I/O virtualization counterparts.

The virtual I/O device model does not need any further hardware or even software support. Hypervisors at each level provide virtual I/O to its own VMs in software, which is transparent to their underlying hypervisors, respectively. Each hypervisor has the benefits from I/O interposition as previously discussed. The downside of this model is poor performance. While using the virtual I/O device model with non-nested virtualization often can deliver sufficient performance with modest overhead, the same virtual I/O device model when used for nested virtualization can result in performance that is many times worse [19] as we observed in the experiments in Chapter 2.

The passthrough model has the best performance [19]. For the guest hypervisor to assign a device to the nested VM, it requires additional hardware or software support beyond what was needed in the case of non-nested virtualization. The host hypervisor needs to have a physical IOMMU to assign a physical device to the guest hypervisor. Furthermore, the host hypervisor needs to provide a virtual IOMMU [6], emulated or paravirtualized, to the guest hypervisor so that the guest hypervisor can then assign the device to the nested VM. In other words, to directly assign a physical device to a nested VM, two levels of IOMMU support are required, the physical IOMMU for the host hypervisor and the virtual IOMMU for the guest hypervisor. The host hypervisor then creates shadow page tables in software by combining mappings in the virtual and physical IOMMUs using the same principles as used for MMUs, which previously did not support two levels of translation. More advanced IOMMU hardware may allow the guest hypervisor to also use a physical IOMMU [5, 57, 13] instead of a virtual one. However, foregoing virtualization features such as migration to use passthrough for improved I/O performance is considered too much of a drawback in many scenarios, especially for cloud computing.

3.2 Virtual-passthrough Design

We introduce virtual-passthrough, a novel yet simple technique for boosting I/O performance when using nested virtualization. Virtual-passthrough is similar to passthrough in allowing a nested VM to directly access the I/O device but assigns virtual I/O devices to nested VMs instead of physical I/O devices. Loosely speaking, virtual-passthrough takes the virtual I/O device model for the host hypervisor and combines it with the passthrough model for subsequent guest hypervisors. The virtual device provided to the guest hypervisor is, in turn, assigned to the nested VM. As shown in Figure 3.2, the nested VM can interact directly with the assigned virtual device, bypassing the guest hypervisor(s).

Unlike the virtual I/O device model, virtual-passthrough avoids the need for guest hy-

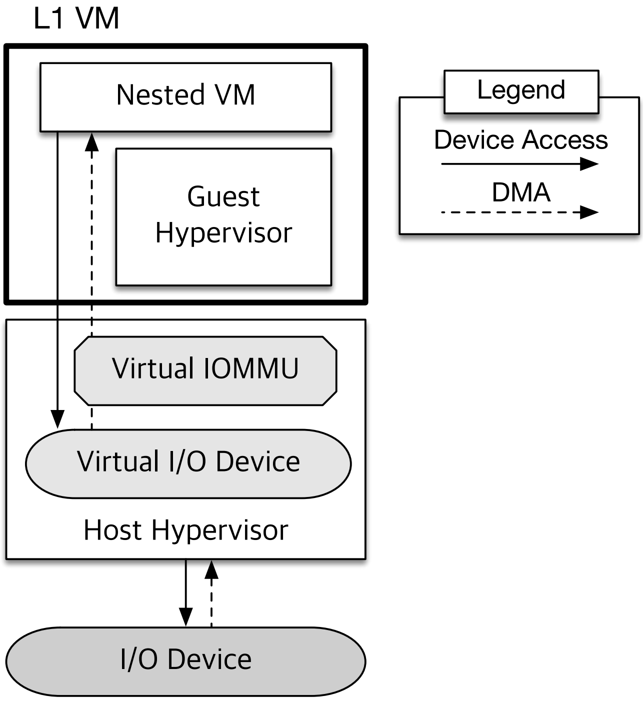


Figure 3.2: Virtual-passthrough

pervisors to provide their own virtual I/O devices, removing expensive guest hypervisor interventions [19] for virtual I/O device emulation. Unlike the passthrough model, virtual-passthrough supports I/O interposition and all its benefits as the host hypervisor provides a virtual I/O device for use by the L1 VM instead of a physical I/O device. For example, it is straightforward to migrate VMs and their nested VMs among different machines. Virtual-passthrough is a software-only solution and does not require any additional hardware. It is easily scalable to support running many VMs on the same hardware for as many virtual I/O devices as desired; no SR-IOV hardware support is required.

Virtual-passthrough is hypervisor and platform agnostic. It works transparently with any virtual I/O device that meets physical device interface specifications such as PCI so that it appears to the guest hypervisors and OSes on any platform just like a physical I/O device. Being hypervisor agnostic is important for cloud computing deployments where various hypervisors are used on servers [26, 95, 16, 84] and, users may freely choose what guest hypervisors and OSes they want to use. Being platform agnostic is also important as cloud providers move towards diversifying their platforms [18, 90].

3.2.1 System Configuration

Virtual-passthrough requires system configuration changes in how devices are managed and used but requires no additional implementation effort for hypervisors that already support both virtual I/O and passthrough device models. It can be achieved by simply leveraging existing software components already introduced for virtual I/O device model and passthrough model. Virtual-passthrough configures these components in a different way at each virtualization level from the two models but does not need to introduce further components. We discuss how the host hypervisor, guest hypervisor, and nested VM need to be configured to support virtual-passthrough.

Using virtual-passthrough, the host hypervisor provides a virtual I/O device to the guest hypervisor. However, simply using the virtual I/O configuration used for the standard virtual I/O device model is not sufficient. Instead, the host hypervisor must provide virtualized hardware to a VM so that the guest hypervisor running in the VM thinks it has sufficient hardware support for the passthrough model.

Figure 3.3 shows the steps involved for an I/O write operation with virtual-passthrough; what virtual-passthrough does for nested VMs is analogous to what passthrough does for non-nested VMs. In the latter case, passthrough requires the hardware to provide both a physical I/O device to assign as well as a physical IOMMU for translating VM physical addresses to host physical addresses. The VM cannot be directly assigned the physical device without the IOMMU without compromising the safety and isolation guarantees provided by the hypervisor. The hardware ensures that memory accesses from the physical I/O device go through the IOMMU so that the physical I/O device safely accesses the correct memory addresses in the VM. Similarly, virtual-passthrough requires the host hypervisor to provide both a virtual I/O device to assign as well as a virtual IOMMU for translating nested VM physical addresses to VM physical addresses used by the guest hypervisor. Without a virtual IOMMU for the guest hypervisor to use, the guest hypervisor has no mechanism to safely assign the virtual I/O device to the nested VM. With virtual-passthrough, the host hypervisor ensures that memory accesses from the virtual I/O device go through the virtual IOMMU so that the virtual I/O device safely accesses the correct memory addresses in the nested VM. Unlike the passthrough model, virtual-passthrough does not require a physical IOMMU.

Using virtual-passthrough, the guest hypervisor simply assigns the given virtual I/O device directly to the nested VM. What the guest hypervisor does with virtual-passthrough is exactly the same as what it does with the regular passthrough model for nested virtualization. In both cases, the guest hypervisor is given an I/O device and an IOMMU, and if properly configured, the guest hypervisor does not know whether the device or IOMMU are physical or virtual. The guest hypervisor simply unbinds the device from its own device driver and creates mappings in the MMU and IOMMU provided by the underlying hypervisor for direct access between the device and the nested VM. Unlike the virtual I/O device model, the guest hypervisor itself does not provide its own virtual I/O device to the nested VM but simply passes through device access to virtual I/O device provided by the

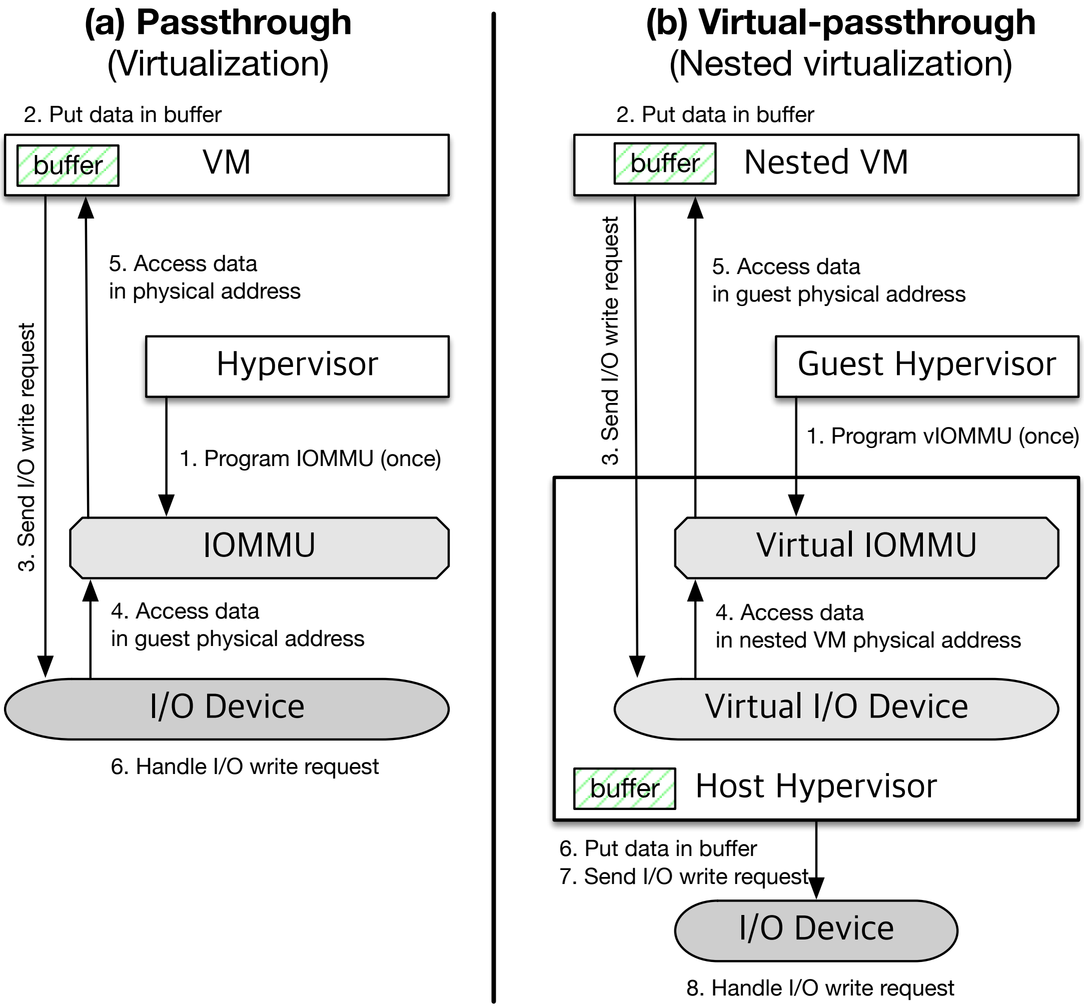


Figure 3.3: I/O Write Operation with (Virtual) Passthrough

host hypervisor.

Any guest hypervisor that provides support for passthrough can use virtual-passthrough. However, since the passthrough model was developed for physical I/O devices, most hypervisor implementations expect the I/O devices used with the passthrough framework to conform to physical device interface specifications, the most common of which is PCI. While virtual I/O devices, especially paravirtual I/O devices, may use any device interface as a software-only solution, those that do not adhere to a standard physical device interface specification are likely to not be assignable or work properly with existing passthrough implementations. However, PCI-based virtual I/O devices are widely available and are assignable to work transparently with existing passthrough frameworks to enable virtual-passthrough.

Using virtual-passthrough, the nested VM is directly assigned the virtual I/O device. Just like the passthrough model, this requires that the nested VM have the correct device driver for the given I/O device. Since virtual-passthrough is used with standard PCI devices, the necessary PCI device drivers are widely available for common OSes. The driver may be already included in the OS [67, 29] or can be installed as needed [68, 106, 42]. As the given device just appears as a regular device to the nested VM, the nested VM does not care from where or how the device is given to it or, whether unbeknownst to it, the device is virtual instead of physical. As a result, virtual-passthrough is designed to work transparently with nested VMs without any modifications other than potential device driver installation.

3.2.2 Example

We use an example of sending a network packet from a nested VM to show how virtual-passthrough operates compared to virtual I/O and passthrough models. The send operation can be viewed in three steps: the nested VM asks the device to send data, the device reads the data from the nested VM, and the device sends the data. If the device is a virtual I/O device, then these three steps are repeated again with the next lower virtualization levels until the device is a physical I/O device.

In the first step, the nested VM first prepares data in a buffer that the device can access and then notifies the device to send the data, which is a common operation for all I/O models. How the notification is delivered to the device is different for each I/O model. A notification from the nested VM is essentially a write operation to a device register via MMIO. For virtual I/O, the write causes a trap to the guest hypervisor, which manages the virtual I/O device visible to the nested VM. For passthrough, the device is assigned to the nested VM, so the write is delivered directly to the physical device with no hypervisor intervention. For virtual-passthrough, the write causes a trap to the host hypervisor, which manages the virtual I/O device visible to the nested VM.

In the second step, when the device receives the notification, it reads data from the buffer in the nested VM. The device accesses the data depends on how the device and IOMMU are configured and where they are located in the physical/virtual machines. For virtual I/O, the device reads data from the nested VM without translation, just like physical I/O devices access physical memory directly on a system not having IOMMU. For passthrough, the physical device is located behind the physical IOMMU, so the device accesses the nested VM through the physical IOMMU. For virtual-passthrough, the virtual I/O device is located behind the virtual IOMMU, so the device accesses the nested VM through the virtual IOMMU.

In the last step, the device sends data. If the device is a physical device as for passthrough, then the data is simply sent over the wire. If the device is a virtual I/O device, then we go back to the first step again.

Table 3.1 summarizes all the resulting steps required to send a packet for each of the three I/O models, virtual I/O (V), passthrough (P), and virtual-passthrough (VP). Virtual I/O takes the most number of steps. Furthermore, not all the steps take the same amount of time. The switches between L1 and L2 that occur during steps 3 and 4, as well as steps 15 and 16, are known to be very expensive [19] as we also have shown in Chapter 2, more than an order of magnitude more expensive compared to a simple switch between L2 and L0, or L1 and L0. For nested virtualization with single-level hardware virtualization support as available on modern x86 or Arm servers, trapping to the guest hypervisor may require multiple traps to the host hypervisor before finally switching to the guest hypervisor as known as exit multiplication. Passthrough obviously takes the least number of steps. Virtual-passthrough only takes a few more steps than passthrough and, more importantly, avoids the most expensive steps required by virtual I/O since it bypasses the guest hypervisor. As we show in Section 3.4, this important difference results in virtual-passthrough being able to achieve performance significantly better than virtual I/O and comparable to passthrough.

Steps		**V**	**P**	**VP**
1	L2 puts a packet in the buffer	X	X	X
2	L2 notifies the I/O device	X	X	X
3	Execution switches from L2 to L0	X		X
4	Execution switches from L0 to L1	X		
5	L1 reads the packet from the buffer	X		
6	L1 puts the packet in the buffer	X		
7	L1 notifies the I/O device	X		
8	Execution switches from L1 to L0	X		
9	L0 reads the packet from the buffer	X		X*
10	L0 puts the packet in the buffer	X		X
11	L0 notifies the physical device	X		X
12	Physical device reads the packet	X	X*	X
13	Physical device sends a packet	X	X	X
14	Execution switches from L0 to L1	X		
15	Execution switches from L1 to L0	X		
16	Execution switches from L0 to L2	X		X

Table 3.1: Steps to Send a Packet for Each I/O Model
V, P, and VP means virtual I/O model, passthrough model and virtual-passthrough respectively. X* means the device accesses memory through (v)IOMMU.

3.2.3 Recursive Virtual-passthrough

Virtual-passthrough can be easily used with additional levels of nested virtualization. The configuration of the L0 host hypervisor and the nested VM remain the same as with two levels of virtualization, as discussed in Section 3.2.1. The only difference when using more levels of nested virtualization is how the multiple levels of guest hypervisors are configured, as shown in Figure 3.4.

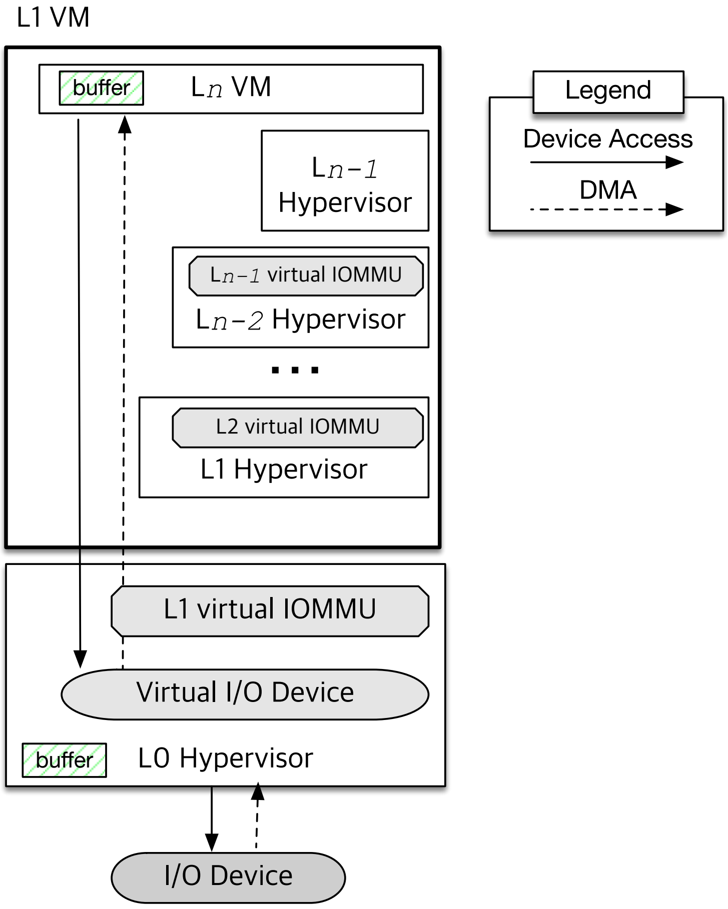


Figure 3.4: Recursive Virtual-passthrough

The guest hypervisors are configured in exactly the same way for recursive virtual-passthrough as they would be for using recursive passthrough. In both cases, the role of the guest hypervisors is to pass through the I/O device, regardless of whether the I/O device is physical or virtual, from the L_k to L_{k+1} VM. For that purpose, each guest hypervisor except the last one provides a virtual IOMMU to the next level hypervisor so that the latter hypervisor can pass through the device to the next level VM. The last level L_{n-1} hypervisor, which is equivalent to the guest hypervisor for two levels of virtualization, only assigns the virtual I/O device to its VM, the L_n VM. The L_{n-1} hypervisor does not need to provide a virtual IOMMU since the VM does not need it to use the assigned I/O device.

Although multiple virtual IOMMUs are needed to configure recursive virtual-passthrough, only the L1 virtual IOMMU is used when the virtual I/O device accesses L_n memory. This is because the L1 virtual IOMMU manages the shadow page tables that contain the combined mappings from L_n VM physical addresses to L1 VM physical addresses. The shadow page tables are built using the same principles as used for building shadow page tables for (non-)recursive passthrough. For example, having more levels of nested virtualization does not change the limited number of steps necessary to send a packet using virtual-passthrough, as shown in Table 3.1. This property is also true for passthrough but is not true for the virtual I/O model. This results in the overhead of using virtual I/O becoming substantially worse than either virtual-passthrough or passthrough with more levels of virtualization.

3.2.4 Migration

Because virtual-passthrough uses I/O interposition with virtual I/O devices, it allows the host hypervisor to encapsulate the state of the L1 VM and decouple it from physical devices to support migration. From the perspective of the host hypervisor, migrating an L1 VM that contains or does not contain a nested VM is essentially the same. The nested VM using virtual-passthrough does not introduce additional hardware dependencies on the host and is completely encapsulated by the host hypervisor. Migration for virtual-passthrough works on any system that already supports migration of VMs that use virtual I/O devices.

The only difference from the perspective of the host hypervisor between virtual-passthrough and virtual I/O is that the former uses a virtual IOMMU while the latter does not. Migration using virtual-passthrough requires that the state associated with the virtual IOMMU is also migrated. However, this is no different than migrating any VM using any other virtual device in which the device state must be properly saved and restored. The virtual IOMMU is software only and is not coupled to any physical device, making it straightforward for the hypervisor to encapsulate its state for migration, just like any other device emulation implementation.

While it is straightforward to migrate a nested VM along with its guest hypervisor without any additional implementation effort, migrating a nested VM alone without the guest hypervisor requires some additional support when using virtual-passthrough. Migration requires transferring the I/O device and VM memory state to the destination. Since copying all memory pages to the destination can take a while, live migration allows a VM to continue executing while the pages are copied, then if some memory pages change, those dirty pages will be re-copied to the destination. When there are not many dirty pages left to re-copy, the VM can be stopped, the remaining dirty pages can be copied over, and the VM can be resumed at the destination, minimizing VM downtime. Migrating a nested VM would be the responsibility of the guest hypervisor, but the challenge when using virtual-passthrough is that the guest hypervisor does not know about what the virtual I/O device is doing because it does not interpose on I/O operations. As a result, the guest hypervisor does not know about the I/O device state that needs to be migrated. Furthermore, since the virtual I/O device can do DMA to the nested VM memory without the guest hypervisor's intervention, the guest hypervisor does not know which pages are dirtied by the I/O device and need to be re-copied to the destination.

We can address this problem by leveraging I/O interposition at the host hypervisor to capture virtual I/O device state and track memory pages dirtied by the virtual I/O device. Whereas capturing device state and tracking dirty pages for physical I/O devices are difficult, the host hypervisor can already do this for virtual I/O devices. The guest hypervisor can then simply ask the host hypervisor to provide it with this information so it can perform the VM migration. All that is needed is to provide an interface between the guest and host hypervisors to deliver the required information about the virtual I/O device and pages dirtied by it, and to modify the guest hypervisor to use this interface instead of disallowing migration because (virtual) passthrough is being used. No modifications are needed to the nested VM.

To provide a standard interface that is hypervisor-independent and device-independent, we leverage the extensibility of the PCI standard, which provides a mechanism known as capabilities to define common functionalities of PCI devices in a standard format. Capabilities allow new functionality to be added to any PCI device and be recognized by system software in a standardized way. Example PCI capabilities include PCI Express and MSI (Message Signaled Interrupts). We define a new PCI device capability, the migration capability, which adds control registers to a virtual I/O device that enables the guest hypervisor to ask the host hypervisor to capture the device state to a specified location and log dirty pages to another specified location. Guest hypervisors that already support PCI devices can then leverage the migration capability in PCI virtual I/O devices to support nested VM migration. By leveraging PCI, any guest hypervisor can interoperate with any host hypervisor. For example, a Xen guest hypervisor can use the migration capability of the virtual device implemented in KVM host hypervisor in a standardized way.

Our approach leverages existing host hypervisor functionality. To save device state, we leave it to the host hypervisor that already has mechanisms to encapsulate its own virtual I/O device state in its own format. The guest hypervisor simply transfers the device state to the destination and does not need to interpret it or understand its format. A caveat with this approach is that it assumes the same type of host hypervisor is used at the source and destination so that the encapsulated state in the source is interpreted correctly in the destination. While specifying a common device state format might enable migrating a nested VM across different host hypervisors, this would be unworkable and overly complicated in practice given the number of different I/O devices and additional hypervisor modifications that would be required.

To track memory pages dirtied by the I/O device, we again can leverage logging functionality that is already implemented by the host hypervisor since it would need to track dirty pages from its own virtual I/O devices for non-nested VM migration. Since the guest hypervisor must interpret the log to determine which pages are dirtied and need to be sent to the destination, we do presume a standardized format for the log, but this is easy to do since it is just a log of addresses. We simply use a bit vector starting at the specified memory location of the log, with each bit representing page. The bit is set if the page is dirtied. The pages are logged based on guest physical addresses for the nested VM, which are conveniently the same memory addresses that the guest hypervisor would use to migrate memory state to the destination, and the same memory addresses that are programmed for use by the virtual I/O device, before translation by the virtual IOMMU, by the guest OS running in the nested VM. This is the same approach already used by KVM/QEMU to track I/O device writes to pages, allowing us to leverage this existing logging functionality without modifications. Because logging is done as part of the existing I/O interposition done by the host hypervisor, it does not require additional traps to the host hypervisor and has minimal impact on performance.

With the new capability, our approach can be easily used with additional levels of virtualization. The L0 hypervisor and the last level L*n-1* hypervisor, need to have the same changes as in the L0 and L1 hypervisor with two levels of virtualization , respectively. All intermediate hypervisors basically don't need any change since they don't use the capability. However, based on the hypervisor implementation, it would require to make the hypervisors aware of the migration capability so that the capability is visible to the next virtualization level as is the case for any new capability. The L*n* VM, the VM being migrated, remains unmodified as before.

Just like we can migrate a VM using virtual I/O model between any virtualization levels, we can also migrate a nested VM using virtual-passthrough between any virtualization levels when hypervisors at each level have aforementioned changes. The nested VM state at any level *n* is completely encapsulated by the L*n-1* guest hypervisor leveraging the migration capability. The only difference between the virtual I/O model and virtual-passthrough is that the latter doesn't support migration between non-nested VM and nested VM. For example, a nested VM using an assigned virtual I/O device can't be migrated to a non-nested VM using a virtual I/O device. In principle, migration is only allowed between VMs created with the same hardware resources, and the assigned device and the virtual I/O device are not the same devices from the hypervisor's perspective. It might be possible to support it with further hypervisor changes such as building virtual I/O device state from the assigned virtual I/O device state and vice versa. Logging is not an issue since it only happens in the source transparent to the destination.

Our solution can be used right away in the current cloud infrastructure with the host and guest hypervisor software update, but the proposed design is not limited to the virtual I/O devices. Since the solution complies with PCI specification, the same capability can be implemented in physical I/O devices. With the same hypervisor changes to use the capability as in the guest hypervisor, it becomes possible to migrate a VM having assigned physical I/O devices.

3.3 Virtual-passthrough Implementation

Virtual-passthrough is designed to be easy to use with existing virtualization infrastructure, making it straightforward to deploy. It requires some system configuration but should require very little if any implementation effort for any system that already has support for device passthrough and virtual IOMMUs. As an example, we describe the system configuration changes and patches needed to use virtual-passthrough with KVM/QEMU hypervisors, Linux guest OSes, and PCI virtual I/O devices.

First, the host hypervisor needs to provide virtualized hardware to the guest hypervisor so that the guest hypervisor thinks it has sufficient hardware support for passthrough.

More specifically, the host hypervisor needs to provide a virtual IOMMU and virtual I/O device, and needs to ensure that memory accesses from the virtual I/O device are going through the virtual IOMMU. For this purpose, we simply use QEMU's architecture-specific IOMMU emulation implementations, specifically Intel VT-d on x86 and Arm SMMUv3 on Arm [14]. The implementations provide support for PCI virtual I/O devices, including architecture-agnostic virtio [92] network devices implemented as PCI devices as specified in Virtio specification [102], and can leverage vhost-net [101, 52], an in-kernel virtio network device implementation in Linux that provides better performance. The IOMMU implementations don't have the direct interrupt injection feature. QEMU patches were required for Arm because the QEMU IOMMU emulation implementations were non-existent in the mainline distribution for Arm.

Second, the guest hypervisor needs to have a way to pass through the virtual I/O device to its nested VM. Linux provides a framework for passthrough, Virtual Function I/O (VFIO) [113]. VFIO is a platform-agnostic framework and exposes PCI and platform devices to userspace [111]. We simply configured VFIO to expose the virtual I/O device to QEMU in the guest hypervisor, which can then program nested VM physical address (PA) to GPA mappings to the virtual IOMMU, and map nested VM PA to the GPA of the device control registers in MMU. The former allows the device to access nested VM's memory, and the latter allows the nested VM to program the device without trapping to the guest hypervisor.

Third, the nested VM needs to have a device driver configured for the assigned virtual I/O device. Any guest OS that supports regular PCI devices is sufficient. For example, Linux running as a guest OS is configured to use virtio PCI devices. As the virtio PCI devices do not appear any different from regular PCI devices, the nested VM OS can discover the assigned virtio device without any problem.

Finally, although virtual-passthrough should support migrating a nested VM along with its guest hypervisor without any further changes, the KVM and QEMU migration implementations were incomplete and did not properly capture virtual CPU state for nested VMs. We applied KVM patches [78] and introduced some additional code in QEMU to fix this problem. To support migrating a nested VM alone, we modified L0 OS and QEMU to implement the PCI migration capability and allow capturing the device state for only a specific device instead of all devices. We then modified the L1 guest OS and L1 QEMU to use the capability. We modified the L1 guest OS to make the PCI migration capability visible to L1 QEMU as the L1 guest OS does not make an otherwise unknown capability visible to userspace, i.e. L1 QEMU, to avoid unexpected usage from userspace. We then modified L1 QEMU to allow migration if all directly assigned I/O devices have the migration capability. the L1 QEMU programs the PCI control registers to obtain the I/O device state and logs memory pages dirtied by the I/O device. The actual device state capture and dirty page logging do not need to be reimplemented since it is already part of L0 OS and QEMU's existing functionality. All changes in the guest hypervisor are completely device-agnostic.

3.4 Experimental Results

We present some experimental results that quantify the impact of virtual-passthrough on nested virtualization performance. We used both x86 and Arm hardware to demonstrate virtual-passthrough works across different hardware platforms. Experiments were conducted using server hardware in CloudLab [41].

x86 measurements were done using Cisco UCS SFF C220 M4 servers, each with two Intel E5-2630 v3 8-core 2.4 GHz CPUs. Hyperthreading was disabled on the nodes to provide a similar hardware configuration to the Arm servers. Each node has 128 GB of ECC memory (8x16 GB DDR4 1866 MHz dual-rank RDIMMs), a 2x1.2 TB 10K RPM 6G SAS SFF HDD for storage, and a Dual-port Cisco UCS VIC1227 VIC MLOM 10 GbE NIC. The x86 hardware includes VMCS Shadowing [55] for nested virtualization, APICv for virtual interrupt support and posted interrupts from CPUs, and VT-d IOMMU support for direct device assignment without the posted interrupts support from devices.

Arm measurements were done using HP Moonshot m400 servers, each with a 64-bit Armv8-A 2.4 GHz Applied Micro Atlas SoC with 8 physical CPU cores. Each m400 node had 64 GB of RAM, a 120 GB SATA3 SSD for storage, and a Dual-port Mellanox ConnectX-3 10 GbE NIC. The Arm hardware includes GICv2 for virtual interrupt support, which has no direct interrupt injection support, and it doesn't have IOMMU. All x86 and Arm servers were connected via 10 GbE, and the interconnecting network switch easily handles multiple sets of nodes communicating with full 10 Gb bandwidth.

To provide comparable measurements, we kept the software environments across all hardware platforms and hypervisors the same as much as possible. All hosts and VMs used Ubuntu 14.04 with the same Linux 4.15 kernel and software configuration unless otherwise indicated. For the host and guest hypervisors, we used KVM with QEMU 2.11.0, with additional patches and modifications to QEMU to correctly support passthrough for nested VMs [122, 123, 121, 112], virtual IOMMUs, and migration, as described in Section 3.3. When using virtual I/O devices with KVM, with or without virtual-passthrough, we used the standard virtio network device with vhost-net and the `cache=none` setting for virtual block storage devices [66, 99, 51]. We also provide measurements using Xen 4.10.1 as an x86 guest hypervisor.

We use the same x86 and Arm hardware as the experiments in Chapter 2 but use up-to-date KVM and QEMU versions. This is to leverage ongoing work, such as IOMMU implementation on Arm, and to apply various bug fixes that are based on the up-to-date software versions.

Because Arm hardware support for nested virtualization is not yet available in hardware [21, 49], we paravirtualize the guest hypervisors to trap to the host hypervisor based on the Arm nested virtualization specification to mimic Arm architectural support for nested virtualization as we discussed in depth in Chapter 2. This is done by essentially replacing instructions that do not trap on existing hardware with hypercalls to trap to the host hypervisor according to the Arm nested virtualization architecture specification. We modified both the host and guest hypervisors to support Armv8.4 nested virtualization, measuring the impact of virtual-passthrough for the upcoming Arm architecture. The guest hypervisor was configured in all cases to run with Virtualization Host Extension (VHE) [30] support.

We used three different configurations for our measurements. We use 4 cores and 12 GB RAM for the native server or VM where we ran workloads. Unless otherwise indicated, we add two more cores and 12 GB more RAM for the hypervisor at each virtualization level: (1) native: running natively on Linux capped at 4 cores and 12 GB RAM, (2) VM: running in a 4-way SMP guest OS with 12 GB RAM using KVM as a hypervisor with 6 cores and 24 GB RAM, and (3) nested VM: running in a 4-way SMP nested guest OS with 12 GB

Netperf	netperf v2.6.0 [60] server running with default parameters on the client in three modes: TCP_RR, TCP_STREAM, and TCP_MAERTS, measuring latency and throughput, respectively.
Apache	Apache v2.4.7 Web server running ApacheBench [100] v2.3 on the remote client, measuring requests handled per second serving the 41 KB file of the GCC 4.4 manual using 10 concurrent requests.
Memcached	memcached v1.4.14 using the memtier benchmark v1.2.3 with its default parameters.
MySQL	MySQL v14.14 (distrib 5.5.41) running SysBench v.0.4.12 using the default configuration with 200 parallel transactions.

Table 3.2: Application Benchmarks

RAM using KVM as the guest hypervisor, which is capped with 6 cores with 24 GB RAM, while the host KVM hypervisor has 8 cores and 36 GB RAM. For benchmarks that involve clients interacting with the server, the client ran on a separate dedicated machine, and the server ran on the configuration being measured, ensuring that clients were never saturated during any of our experiments. Clients ran natively on Linux with the same kernel version as the server and were configured to use the full hardware available. For each architecture, we evaluated performance using widely-used I/O intensive application workloads, as listed in Table 3.2.

For each architecture and system configuration, we considered all possible I/O configurations for network. For native, we used physical I/O devices, the only configuration possible. For VM, we measured both paravirtual I/O and passthrough, the latter only on x86 since the Arm hardware used lacked support for passthrough. For nested VM, we measured paravirtual I/O, passthrough, and virtual-passthrough, with passthrough again only for x86 due to Arm server hardware limitations.

Figure 3.5 shows performance measurements on x86 for five different VM configurations. Since we are more interested in overhead rather than absolute performance, VM and

Application	Unit	Baseline	L1	L2	L2 + VP	L3	L3 + VP
Netperf RR	trans/sec	39,478	27,217	8,666	10,691	537	534
Netperf Stream	Mbits/sec	9,413	9,413	9,401	9,410	2,258	9,239
Netperf Maerts	Mbits/sec	9,412	9,413	5,051	9,407	N/A	9,176
Apache	trans/sec	18,202	11,957	4,418	9,814	268	1,170
Memcached	trans/sec	424,321	424,504	147,070	418,126	4,141	91,234
MySQL	sec	7.0	14.2	18.0	16.2	64.5	35.8

Table 3.3: Application Benchmark Raw Performance on x86
(VP stands for virtual-passthrough.)

nested VM performance are normalized relative to x86 native execution, with lower meaning less overhead. Table 3.3 shows the non-normalized results from the measurements. Note that the numbers are slightly different from Table 2.9 because of differences in the Linux kernel and QEMU versions used.

For the VM case, both paravirtual I/O and passthrough provide mostly similar performance, with passthrough having somewhat better performance for both Netperf TCP_RR and Apache. The virtual I/O device model overall provides sufficient performance for the VM case so that passthrough provides only marginal gains for these application workloads. One workload, MySQL, has noticeable overhead using paravirtual I/O, but using passthrough instead provides no real performance benefit.

For the nested VM case, the performance differences among the different I/O configurations are substantial. Paravirtual I/O performs significantly worse than passthrough for the nested VM case for most of the application workloads, more than 3 times worse than the VM case for Apache and Netperf TCP_RR. In contrast, virtual-passthrough delivers nested VM performance comparable to passthrough and almost as good as the VM case for all application workloads except Netperf TCP_RR. The performance gains of using virtual-passthrough instead of the virtual I/O device model are substantial, more than doubling

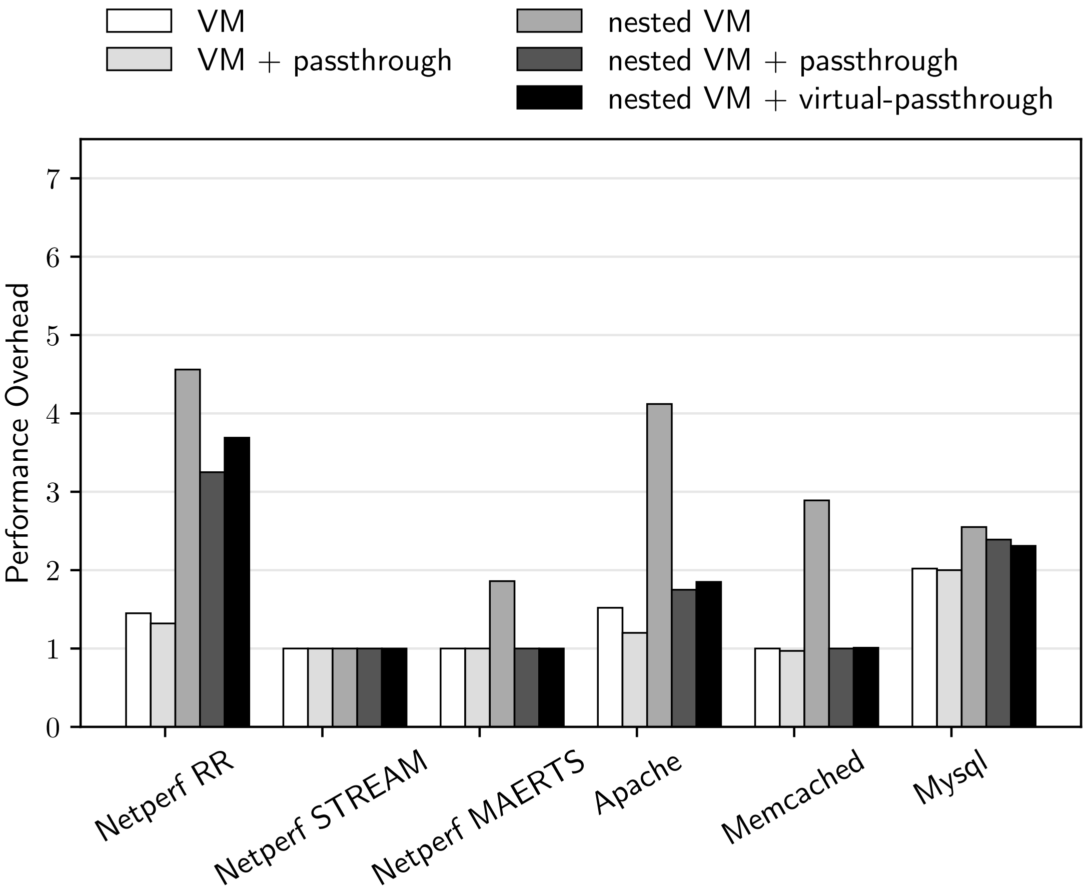


Figure 3.5: Application Performance on x86

performance for Apache and almost tripling performance for Memcached.

The one workload that virtual-passthrough does not help that much is Netperf TCP_-RR, in which only one byte of data is transferred back and forth between the server and the client. Passthrough does not help much either. The reason is that this is a latency-sensitive workload that does not involve much data transfer and results in the VM and nested VM idle waiting for responses from the client and therefore going to sleep. Waking up the nested VM from the idle state on responses involves expensive switches between the guest hypervisor and the nested VM, as described in steps 14 to 16 in Table 3.1 but for all I/O configurations. Direct interrupt injection to the nested VM does not mitigate this problem because that only helps when the nested VM is running. Virtual-passthrough and passthrough show somewhat better performance than paravirtual I/O since the send operation is cheaper, as discussed in Section 3.2.2.

TCP_RR has another source of overhead on x86 for all I/O configurations. By default, the Linux kernel runs using tickless mode [63], which makes idle CPUs not receive periodic timer interrupts (i.e., ticks), to save power. This involves programming the timer when the kernel enters and exits the idle state [79]. This is on the critical path because the timer is programmed when the VM exits the idle state after getting an interrupt for an incoming network packet. Accessing the timer register to program the timer is a simple and cheap operation on native machines, but is emulated by KVM, making it a very expensive operation for nested VMs. We confirmed that changing the kernel to use periodic tick mode, thereby avoiding timer programming when exiting from the idle state, greatly reduces overhead for all I/O configurations.

Figure 3.6 shows the same performance measurements on x86 as Figure 3.5, but with one more level of nesting by running the nested virtualization workloads in an L3 VM. These measurements show that adding an additional level of nesting makes paravirtual I/O performance much worse overall compared to passthrough, but virtual-passthrough continues to offer similar performance to passthrough even with an additional level of nesting.

Figure 3.7 shows the same performance measurements on x86 as Figure 3.5, but using Xen instead of KVM. However, because nested virtualization support does not work properly in recent Xen versions, including the version we used [116], we ran Xen only as the guest hypervisor for the nested VM cases while using KVM as the host hypervi-

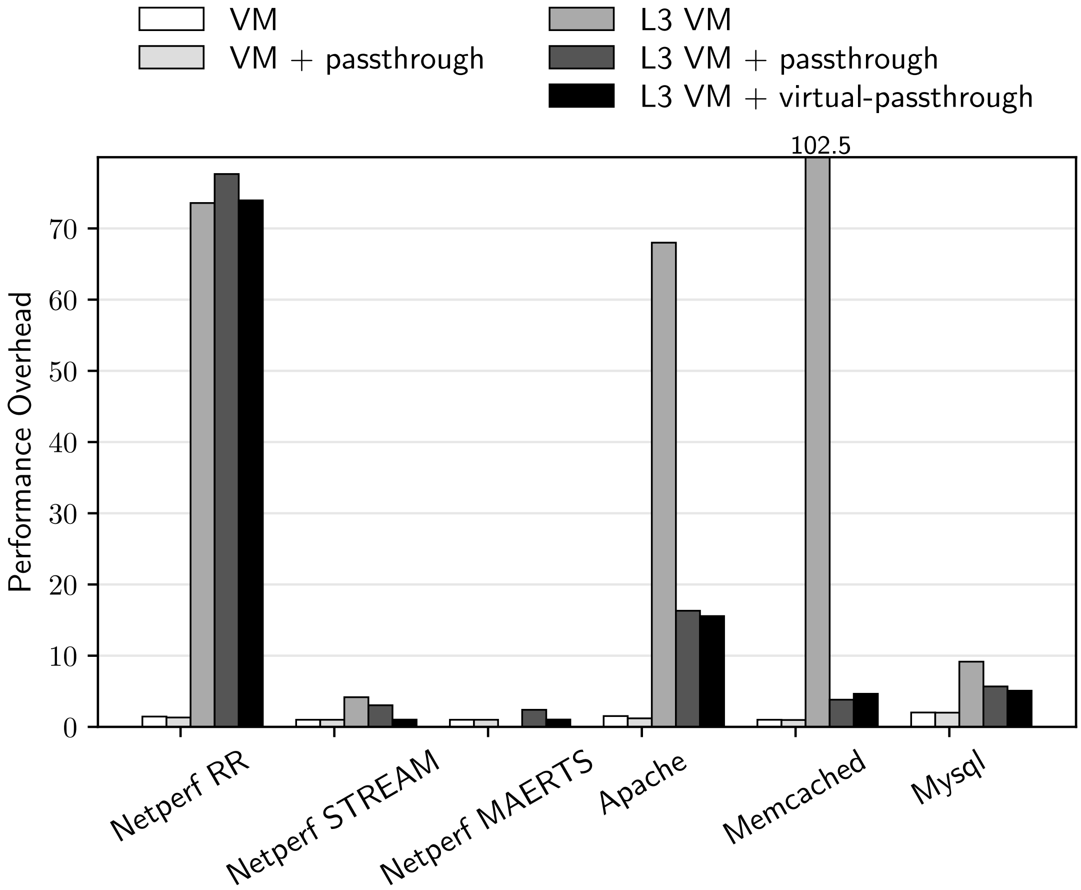


Figure 3.6: Application Performance on x86 in L3 VM

sor. Just as in Figure 3.5, the performance differences among the different I/O configurations are substantial for the nested VM case. Paravirtual I/O performs significantly worse than passthrough for the nested VM case for most of the application workloads. Virtual-passthrough is able to provide performance similar to passthrough for all workloads and provide substantial gains over the virtual I/O device model, almost doubling performance for Apache and quadrupling performance for Memcached. For MySQL, virtual-passthrough does not provide much performance gain over the virtual I/O device model, but its performance is still similar to passthrough, so the limited performance benefit here is due to the limited gains of using passthrough with Xen. A surprising result is

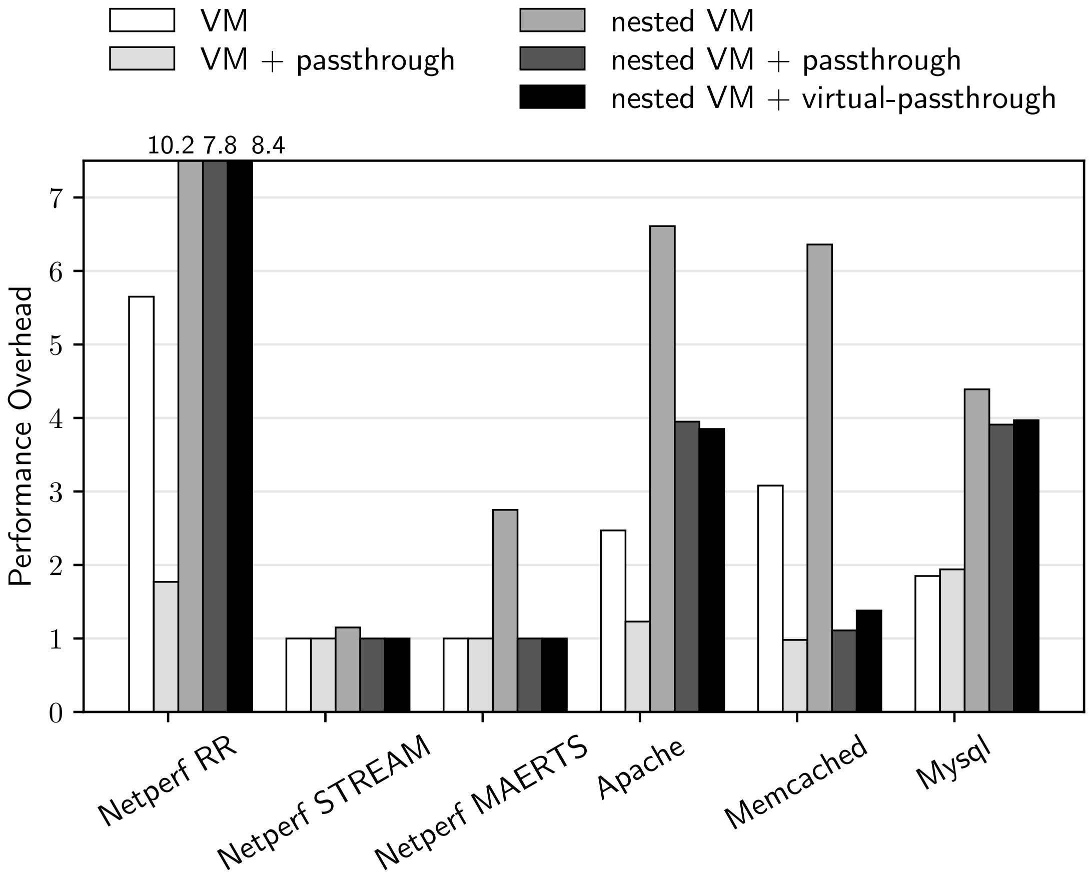


Figure 3.7: Application Performance on x86, Xen on KVM

that virtual-passthrough provides even better performance than the non-nested VM case for Memcached. For the VM case, Figure 3.7 shows that Xen's paravirtual I/O device does not perform as well as passthrough, with substantial overhead for Netperf TCP_RR, Apache, and Memcached. This is in contrast to the good paravirtual I/O performance in the VM case for KVM in Figure 3.5. With virtual-passthrough, KVM is the host hypervisor so the nested VM running on the Xen guest hypervisor is using KVM's paravirtual I/O device. The nested VM, therefore, benefits from the better performance of KVM's paravirtual I/O device and outperforms the non-nested VM using Xen's paravirtual I/O device.

Figure 3.8 shows performance measurements for three different VM configurations on

Application	Unit	Baseline	L1	L2	L2 + VP
Netperf RR	trans/sec	26,754	15,486	4,565	7,171
Netperf Stream	Mbits/sec	9,410	8,488	6,478	9,146
Netperf Maerts	Mbits/sec	9,403	9,407	8,211	7,941
Apache	trans/sec	5,958	5,546	2,534	3,294
Memcached	trans/sec	160,222	138,812	35,774	83,753
MySQL	sec	13.9	17.9	25.5	25.3

Table 3.4: Application Benchmark Raw Performance on Arm
(VP stands for virtual-passthrough.)

Armv8.4, respectively, with performance normalized to Arm native execution, lower meaning less overhead. Table 3.4 shows the non-normalized results from the measurements. Note that the numbers are slightly different from Table 2.9 because of differences in the Linux kernel and QEMU versions used.

Unlike x86, blank measurements are shown for passthrough configurations since it was not possible to run them on Arm due to a lack of hardware support. In contrast, virtual-passthrough measurements show good performance on Arm despite the lack of IOMMU hardware. For the VM case, performance is generally quite comparable to native execution with only modest overhead, indicating that the virtual I/O device model overall provides good enough performance, just as with x86. Normalized performance is actually somewhat better than x86, as MySQL no longer incurs much overhead when comparing VM versus native execution.

Although we showed in Chapter 2 that NEVE significantly improves the performance of nested virtualization compared to Armv8.3, it still incurs high overhead for some application workloads, including Memcached that runs roughly four times slower for the nested VM versus the VM case when using paravirtual I/O. Virtual-passthrough reduces the performance overhead in all cases even further, including reducing the overhead of

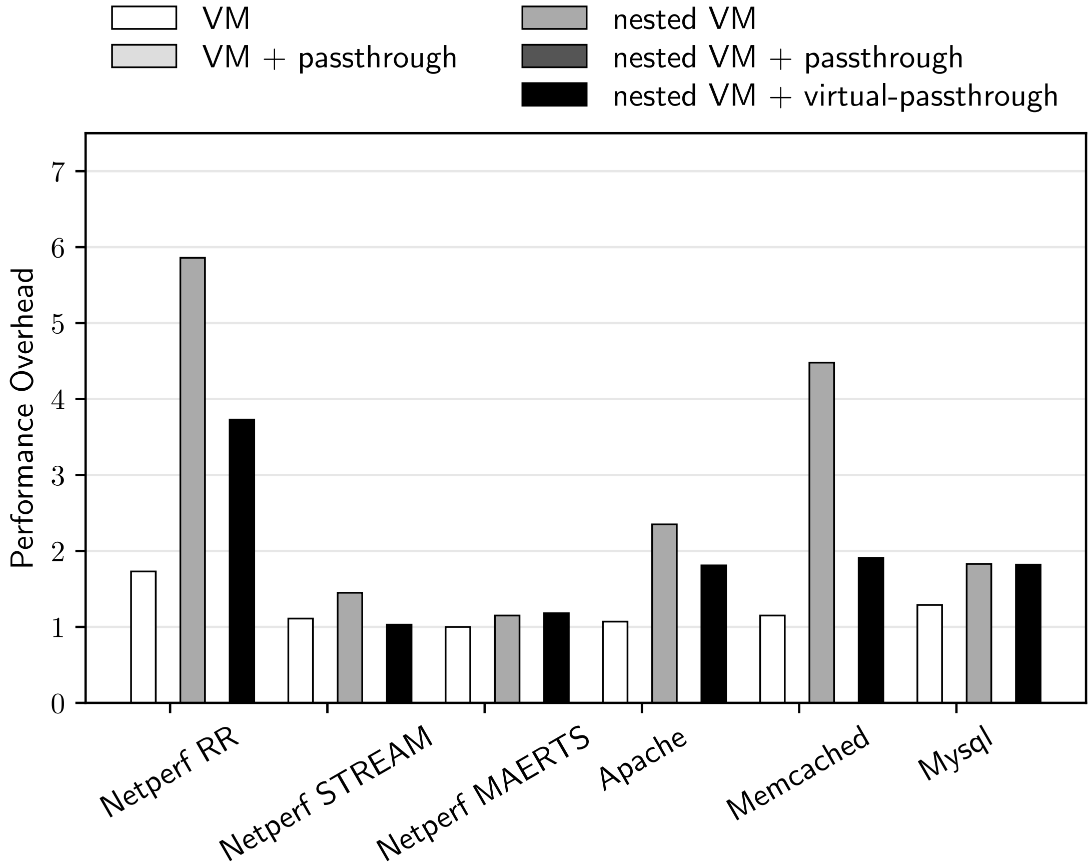


Figure 3.8: Application Performance on Arm

Memcached by more than two times. The one case in which performance overhead still remains high is with Netperf TCP_RR, just like for x86 and for the same reasons. Nevertheless, these measurements across multiple architectures show that virtual-passthrough can make a significant performance improvement for nested virtualization.

Figure 3.9 and 3.11 show the total migration time for VM and nested VM respectively in seconds for running the same workloads on x86 as used in Figure 3.5. Live migration was done between two identical x86 servers on the same subnet. The default transfer bandwidth configuration was used for QEMU for migration, which was 268 Mbps, to avoid interference with the running workload.

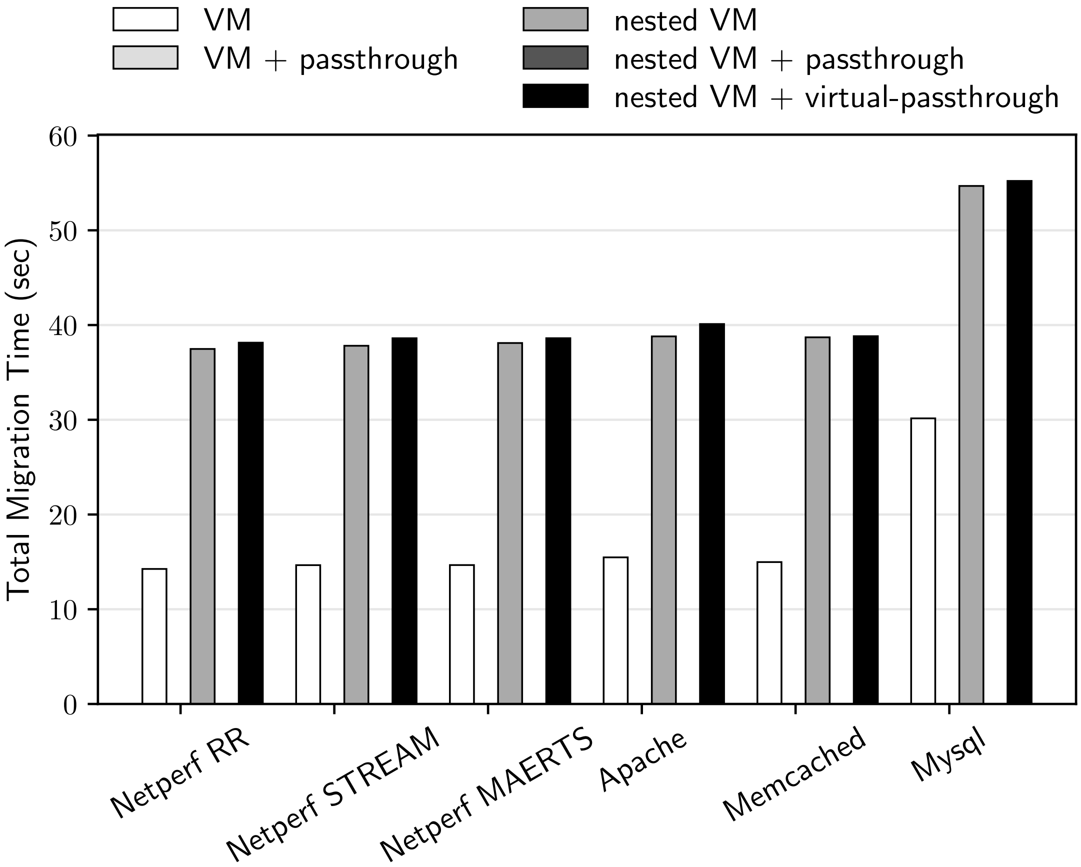


Figure 3.9: Total VM Migration Time on x86

Figure 3.9 shows that both paravirtual I/O and virtual-passthrough provide similar migration performance in terms of total migration time, while none of the passthrough configurations provide any migration capability, as represented by the blank measurements for those configurations. We confirmed that total migration time was bandwidth limited and can be reduced further by increasing the QEMU transfer bandwidth configuration.

Figure 3.9 shows that migration with the VM configuration is faster than with the nested VM configuration. This is primarily because we added more memory for each virtualization level, so there is substantially more memory state to migrate with the nested VM configuration. There is also more state to migrate because we added more cores for each

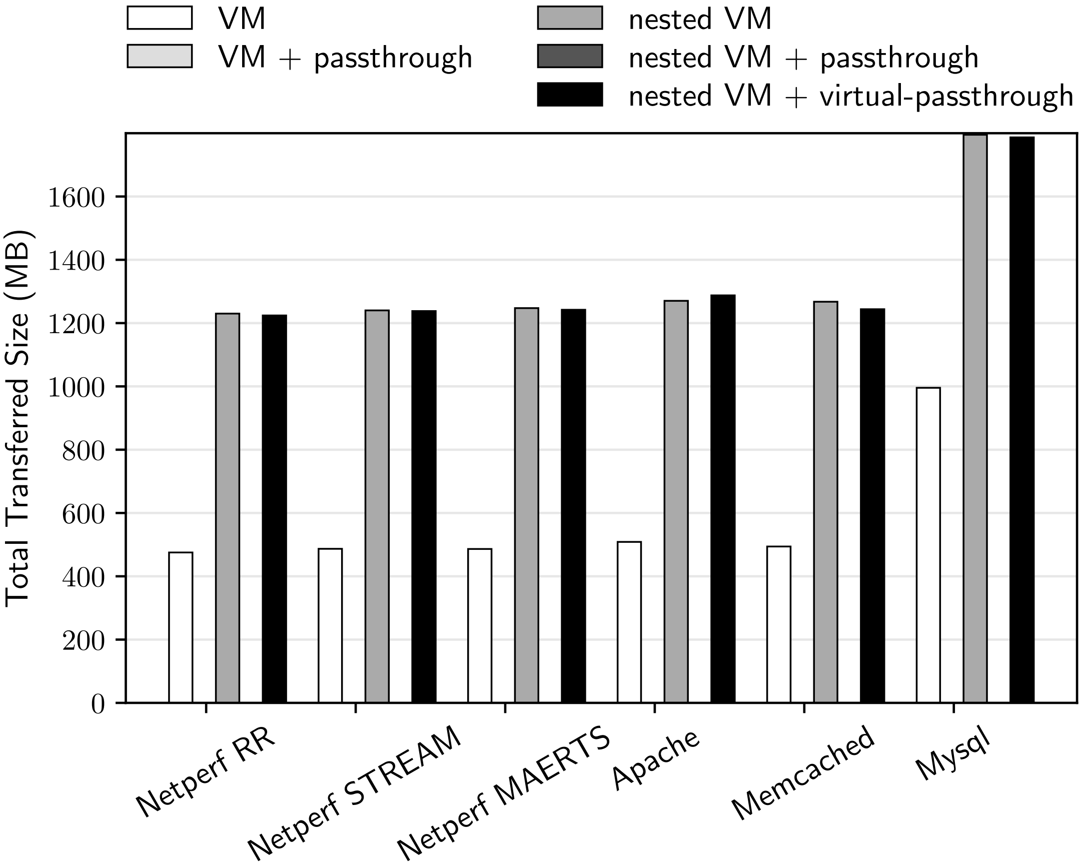


Figure 3.10: Total Transferred Size on x86

virtualization level, resulting in more CPU state, and running the workload in the nested VM on top of the VM results in more dirty pages. Figure 3.10 shows the transferred size in megabytes during migration for each measurement in Figure 3.9. Nested VM configurations indeed have more data to transfer compared to the VM configuration. Changing the VM virtual hardware configuration to have the same total memory and number of cores as the nested VM configuration would result in VM migration times much closer to nested VM migration times.

Most of the workloads have similar migration times, except for MySQL, because it creates many dirty pages due to caching database changes before flushing them to disk.

Although total migration time requires tens of seconds in all cases, actual downtime of the VM being migrated ranged between 15 and 35 ms for both paravirtual I/O and virtual-passthrough, showing that migration can be done with very little impact on application availability.

Figure 3.11 shows the VM migration times for just migrating the nested VM without migrating the underlying VM. For comparison, Figure 3.11 also shows the VM migration time for the non-nested VM configuration from Figure 3.9. Nested VM migration configurations with paravirtual I/O and virtual-passthrough show similar migration times, while the empty bars show that migration does not work with the passthrough configuration. In contrast to Figure 3.9, the migration times for nested VM configurations are similar to the VM migration time because migrating the nested VM no longer involves also migrating the underlying VM. As a result, the amount of state migrated is similar, with similar amounts of virtual CPU and memory state. Similar to the non-nested VM migration results, the downtime for migrating the nested VM took less than 35 ms for both paravirtual I/O and virtual-passthrough. These results show that virtual-passthrough provides a unique combination of superior I/O performance for nested virtualization while supporting migration, a key virtualization feature for cloud computing deployments.

3.5 Related Work

Much work has been done on analyzing and improving the I/O performance of VMs, including using I/O emulation [98], paravirtualization [80, 94, 88, 47, 120, 50, 81, 53, 64], and direct device assignment [77, 115, 87, 114, 46, 7, 103, 38, 76, 31]. While none of these

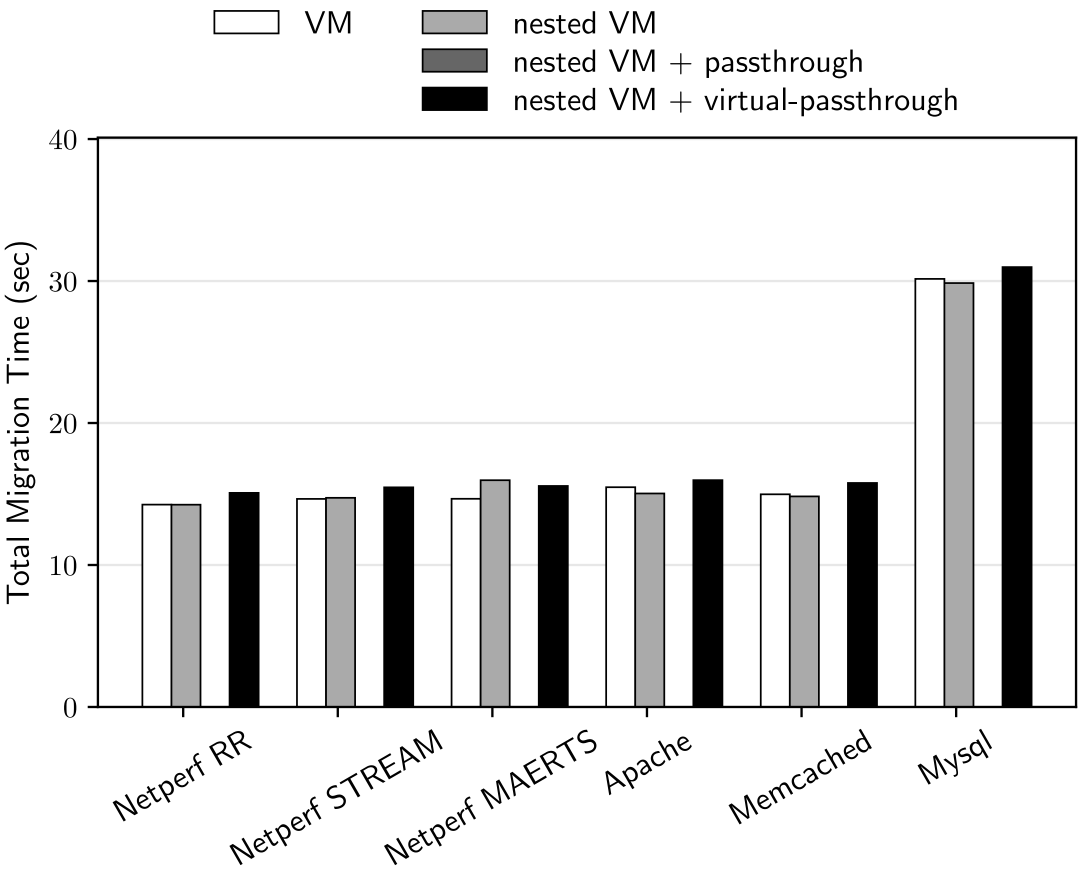


Figure 3.11: Total Nested VM Migration Time on x86

proposed optimizations is specifically for nested virtualization, they are complementary to our approach and, in some cases, can be used to further optimize I/O performance when using virtual-passthrough.

vIOMMU [6] was the first to evaluate non-nested VM performance using a virtual IOMMU on x86. They introduced using virtual IOMMUs along with the physical IOMMU to provide protection for VMs running unmodified guest OSes when using direct device assignment. A virtual IOMMU was exposed to the guest OS to provide a virtualized view of the physical IOMMU, and various techniques were explored to optimize its performance. Virtual-passthrough leverages virtual IOMMUs but for a completely different purpose, assigning virtual devices to a nested VM, and doing so without requiring a physical IOMMU. The virtual IOMMU does not need to be exposed to the nested VM for this purpose and mappings are created only once when launching the nested VM, avoiding the performance costs associated with direct control of the IOMMU by the guest OS. It is still possible to expose the virtual IOMMU if desired to the nested VM, in which case the optimizations presented would help performance for virtual-passthrough.

Turtles [19] mentions that nested VM I/O support can be done in nine possible combinations of emulation, paravirtualization, and direct device assignment by picking any approach for I/O virtualization between host hypervisor and VM, and between guest hypervisor and nested VM. They evaluated the combinations they considered interesting with device passthrough performing the best but did not recognize the idea or benefits of directly assigning virtual devices to a nested VM, as we introduce with virtual-passthrough. We show for the first time the power of this previously dismissed approach, its ability to provide performance comparable to direct physical device assignment for many I/O workloads without requiring additional hardware support, and its ability to provide I/O interposition benefits such as migration.

We have introduced NEVE, hardware extensions on Arm to improve nested virtualization performance, in Chapter 2. While I/O application performance improved by an order of magnitude over Armv8.3, the end result showed that overhead is still significant compared to bare-metal performance when using paravirtual I/O. Our results show that virtual-passthrough can further improve Arm nested virtualization performance such that many application workloads are now much closer to native Arm performance.

Various efforts have tried to compensate for the lack of I/O interposition with passthrough to support live migration. Most of the previous works took software-only approaches [85, 61, 124, 125] since changing hardware is often not easy or feasible. Without hardware support, software-only mechanisms are indirect and complex such as keeping monitoring the device ring buffer with a high frequency to identify dirty pages. Those software-only approaches either do not support unmodified guest OSes or may lose data due to incomplete tracking of I/O operations. Our approach, however, leverages the virtual hardware capability, which is easier to add compared to the physical counterpart, to simply and directly get the device state and the dirty pages. No nested VM change is required.

There are approaches proposing new hardware functionalities. ReNIC [36] proposed to extend SR-IOV device functionality for device state migration and IOMMU functionality for dirty page logging. Despite the similarities to our work leveraging hardware extensions, there are at least three key differences. The migration capability in our design is formally defined hence can be applied to any PCI I/O devices. On the other hand, ReNIC hardware extensions are limited to the SR-IOV capable devices, which is not the case for many virtual I/O devices, while it's also not clear how ReNIC SR-IOV specific extensions can be applied to different I/O devices. Secondly, we implemented the device extensions in the exactly the same device that the nested VM used to use, the virtual I/O device, for the realistic evaluation while ReNIC hardware extensions are emulated in CPUs instead of implemented in the I/O device, which resulted in less realistic evaluations as only a fraction of the full bandwidth of the underlying I/O device is used. Lastly, the dirty page tracking mechanism is different. We added the dirty page logging functionality to the I/O device while ReNIC did it to IOMMU. While it is possible to log dirty pages in both devices, doing it in IOMMU may slow down the I/O performance during migration. For example, to log dirty pages, IOMMU forces the I/O devices to request the address translation for each access, which otherwise can be cached in the I/O device for the faster translations [86] while the I/O device still can log the accesses.

vDPA(vhost Data Path Acceleration) [72] introduced a hybrid approach for virtio devices [92] to keep the control plane in the hypervisor for I/O interposition as before but to offload the data plane to the hardware accelerator for enhanced performance. Using vDPA alone for nested virtualization with virtual I/O model, however, wouldn't improve the performance much. The performance bottleneck, the nested VM using the virtual device provided by the guest hypervisor, still exists while the guest hypervisor may get better virtual I/O performance with vDPA. Using the same configuration described in Section 3.2.1, vDPA can be used with virtual-passthrough for the enhanced performance. Since using vDPA is transparent to the guest hypervisor allowing no I/O interposition as other assigned devices, adding the proposed migration capability to the virtual I/O device with vDPA helps the guest hypervisor to support the nested VM migration using virtual-passthrough.

To support VM migration running applications using DPDK [39], the userspace networking stack, DPDK allows user applications to directly use a selected set of virtual I/O devices instead of physical I/O devices [40]. It shares some similarities to virtual-passthrough in that virtual I/O device is directly accessible from a less privileged CPU mode in a VM, but there are significant differences. First, using DPDK requires rewriting applications to use poll mode drivers while virtualization supports unmodified software stacks in a VM, which is the whole point of using virtualization in the first place. Second, DPDK would not perform well for additional levels of nested virtualization without the technique we propose. For example, when running DPDK with the virtual I/O device in the nested VM without virtual-passthrough, it would suffer from a performance drop since the given device is emulated by the guest hypervisor. With virtual-passthrough, however, the DPDK application in nested VM can use the virtual device provided by the host hypervisor, which would help to achieve much better performance like showed in Section 3.4. Lastly, DPDK does not allow multiple applications in a VM to use the same device. Running multiple DPDK applications typically involves running one VM per each application, which requires more management effort, and is not very scalable. Our approach only needs to launch one nested VM with one virtual device to run an arbitrary number of applications in it.

3.6 Summary

We introduced virtual-passthrough, a novel yet simple technique for boosting I/O performance when using nested virtualization. Virtual-passthrough is similar to direct physical device assignment but instead assigns virtual I/O devices to nested VMs. Virtual devices provided by the host hypervisor can be assigned to nested VMs directly without delivering data and control through multiple layers of virtual I/O devices. Therefore, virtual-passthrough reduces exit multiplication by eliminating the need for guest hypervisor execution when the nested VM interacts with the assigned virtual I/O devices. The approach leverages the existing direct device assignment mechanism and implementation, so it only requires virtual machine configuration changes. Virtual-passthrough preserves I/O interposition in the host hypervisor different from physical device passthrough while virtual-passthrough also can easily support important I/O interposition benefits such as migration in the hypervisors at intermediate layers. Scalability is not a problem as many virtual devices can be supported by a single physical device. Supporting both paravirtual and emulated I/O devices is straightforward. The technique is platform agnostic, does not require hardware support such as physical IOMMUs or SR-IOV. We have applied virtual-passthrough in KVM for both x86 and Arm hardware, and show that it can provide more than an order of magnitude improvement in performance over current KVM virtual device support on real application workloads.

Chapter 4

Optimizing Nested Virtualization Performance Using Direct Virtual Hardware

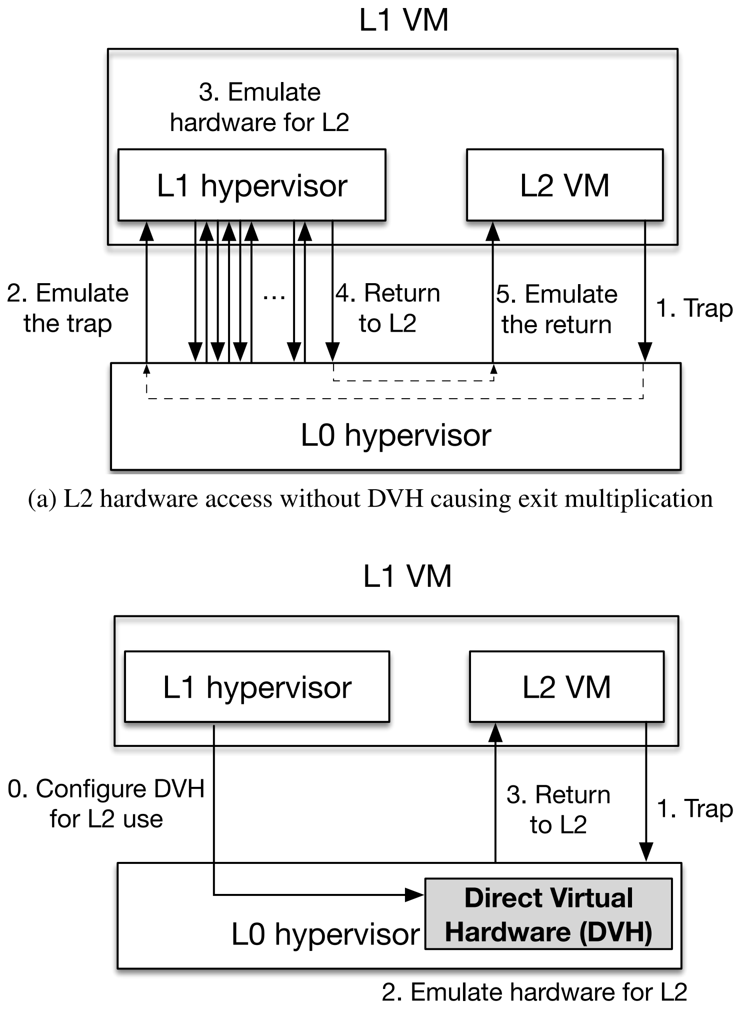
In Chapter 3, we presented virtual-passthrough to improve performance of I/O intensive workloads running in nested VMs by eliminating the need for guest hypervisor execution when the nested VM interacts with the assigned virtual I/O devices. While we show that virtual-passthrough is effective for many application workloads, there remain many applications that still do not perform well compared to native execution, such as latency-sensitive applications. Furthermore, virtual-passthrough does not help much for applications that are not I/O intensive.

To address this problem, we introduce Direct Virtual Hardware (DVH), a new approach that enables a host hypervisor to directly provide virtual hardware to nested VMs without the intervention of multiple levels of hypervisors. DVH is a generalization of virtual-passthrough and does not limit virtual hardware to I/O devices. The virtual hardware appears to intervening layers of hypervisors as additional hardware capabilities provided by the underlying system even though, in actuality, the capabilities are provided by the host hypervisor in software. Like virtual-passthrough, DVH reduces exit multiplication by eliminating the need for guest hypervisor execution when the nested VM accesses the virtual hardware. DVH makes it possible to support novel virtualization optimizations only in software, and even introduce new virtual hardware capabilities that are not natively supported by hardware. Like other real hardware mechanisms, virtual hardware requires guest hypervisors to be aware of these capabilities to use them but is transparent to nested VMs. DVH can be realized on a range of different architectures. Beyond virtual-passthrough, we present three additional DVH mechanisms: virtual timers, virtual inter-processor interrupts (IPIs), and virtual idle.

We have implemented DVH in the Linux KVM hypervisor and evaluated its performance. Our results show that combining the four DVH mechanisms can provide even greater performance than virtual-passthrough alone and provide near-native execution speeds on real application workloads even for multiple levels of recursive virtualization. We also show that DVH can provide better performance than device passthrough while at the same time enabling migration of nested VMs, thereby providing a combination of both good performance and key virtualization features not possible with device passthrough.

4.1 Design

DVH mitigates the exit multiplication problem of nested virtualization by having the host hypervisor directly provide virtual hardware to nested VMs, which reduces the need for forwarding nested VM exits to the guest hypervisor. Virtual hardware appears to guest hypervisors as additional hardware capabilities provided by the underlying system, even though the virtual hardware is in actuality provided in software by the host hypervisor. Because guest hypervisors don't need to use virtual hardware for their own execution, nested VMs can be allowed to access, configure, and manipulate virtual hardware without the need



(a) L2 hardware access without DVH causing exit multiplication

(b) L2 hardware access with DVH

Figure 4.1: Hardware Access from Nested VM

to exit to guest hypervisors for emulating the respective hardware behavior as shown in Figure 4.1b. DVH is designed to be transparent to nested VMs. The host hypervisor maps the virtual hardware to what the nested VM perceives is the physical hardware, requiring no changes to nested VMs.

Directly providing virtual hardware to VMs does require exits from the VM to the host hypervisor because virtual hardware is not real hardware, so the host hypervisor needs to emulate the hardware behavior for the VM. DVH, therefore, trades exits to guest hypervisors for exits to the host hypervisor. For non-nested virtualization, DVH provides no real benefit because it still requires exits to the hypervisor. However, for nested virtualization, the potential benefit is significant because exits to just the host hypervisor are much less expensive than exits to guest hypervisors. On modern hardware with single-level architectural support for virtualization, all exits always go first to the host hypervisor. If the exit needs to be handled by a guest hypervisor, the host hypervisor then forwards the exit to the guest hypervisor. Fundamentally, an exit to a guest hypervisor is more expensive than an exit to the host hypervisor by at least a factor of two because it also requires at least one exit to the host hypervisor. In practice, an exit to a guest hypervisor is much more expensive than a factor of two because it often requires many additional exits to the host hypervisor to perform guest hypervisor's operations that are not allowed to execute natively. By trading potentially many exits due to switching to guest hypervisors for one exit to the host hypervisor, DVH can potentially bring the cost of nested virtualization down to non-nested virtualization, in which exit multiplication does not exist.

DVH differs from previous approaches such as a hypervisor providing virtual hardware to its guests or architecture extensions for nested virtualization. In the first approach, the hypervisor providing the virtual hardware is the same as the hypervisor responsible for managing the VM itself. In contrast, DVH provides virtual hardware from a hypervisor layer different from the one responsible for managing the VM, thereby providing the hypervisor managing the VM with an abstraction that appears to be real hardware. For nested virtualization, DVH gains its advantages by providing virtual hardware directly from the host hypervisor, not from the guest hypervisor. Unlike previous approaches, DVH provides virtual hardware directly to the nested VM, so there is no longer a need to exit to the guest hypervisor. In the second approach, which includes VMCS shadowing on x86 [55] and NEVE on Arm introduced in Chapter 2, architecture extensions defer unnecessary traps from the guest hypervisor, resulting in less number of traps to the host hypervisor in step 3 in Figure 4.1a. However, the number of exits from nested VMs to the guest hypervisor, which is the root cause of the nested virtualization overhead, does not change. In contrast, DVH directly addresses the root cause and reduces the number of exits from the nested VM to the guest hypervisor. This completely removes steps 2, 3, and 4 in Figure 4.1a when virtual hardware is supported. Architectural support for nested virtualization and DVH are complementary, optimizing different aspects of nested virtualization. For cases where DVH cannot avoid exiting to the guest hypervisor, for example, due to a hypercall from a nested VM, the architectural support can help to reduce overhead.

DVH provides at least two other benefits for nested virtualization. First, it preserves the host hypervisor's ability to interpose on virtual hardware accesses, allowing it to transparently observe, control, and manipulate those accesses. Second, because virtual hardware is just software, it is not limited by physical hardware. Virtual hardware can be designed to be the same as an existing physical hardware specification, regardless of the existence of the physical hardware on the system. Virtual hardware can also be designed to extend the existing hardware to provide more powerful and efficient hardware to the VMs. No physical hardware support is required.

While the guest hypervisor no longer needs to emulate hardware accesses from nested VMs with DVH, it does need to configure and manage the virtual hardware. The guest hypervisor needs to check if virtual hardware is available on the system, and configure the virtual hardware for use by nested VMs, as shown in step 0 in Figure 4.1b. An important aspect of the guest hypervisor's configuration is to enable the host hypervisor to obtain any information it needs from the guest hypervisor to emulate the virtual hardware for the nested VM. This can include information internal to how the guest hypervisor manages its nested VM, which would not be accessible to the host hypervisor unless it is provided by the guest hypervisor. The information can be passed to the host hypervisor via either existing architectural support for virtualization or new virtual hardware interfaces designed for this purpose.

DVH is essentially a system design concept, which can be applied to and realized on different architectures with single-level virtualization hardware support. We introduce several DVH mechanisms for the x86 architecture, as discussed in Sections 4.1.1 to 4.1.4. DVH can be easily used with additional levels of nested virtualization and supports key virtualization features such as live migration, as discussed in Sections 4.1.5 and 4.1.6.

4.1.1 Virtual-passthrough

Virtual-passthrough, as we discussed in Chapter 3, is a technique for a nested VM to transparently interact with virtual I/O devices provided by the host hypervisor without the interventions from guest hypervisors. Virtual-passthrough is, in fact, a perfect example of the DVH design; the host hypervisor provides additional virtual I/O devices, and the guest hypervisor passes the additional virtual I/O devices to nested VMs for direct access, as shown in Figure 4.1b.

4.1.2 Virtual Timers

Guest OSes in VMs make use of CPU hardware timers that can be programmed to raise timer interrupts, such as the local Advanced Programmable Interrupt Controller (LAPIC) timer built into Intel x86 CPUs. Because the LAPIC timer may also be used by hypervisors, when the guest OS programs the timer, this causes an exit to the hypervisor to emulate the timer behavior. Emulation can be done by using software timer functionality, such as Linux high-resolution timers (hrtimers), or by leveraging architectural support for timers, such as the VMX-Preemption Timer that is part of Intel's Virtualization Technology (VT). For nested virtualization, the guest hypervisor is responsible for emulating the timer behavior for a nested VM. However, because of exit multiplication, exiting to the guest hypervisor to emulate the timer behavior is expensive.

We introduce virtual timers, a DVH technique for reducing the latency of programming timers in nested VMs. A per virtual CPU virtual timer is software provided by the host hypervisor that appears to guest hypervisors as an additional hardware timer capability. For example, for x86 CPUs, the virtual timer appears as an additional LAPIC timer so that guest hypervisors see two different LAPIC timers, the regular LAPIC timer and the virtual LAPIC timer. Like the LAPIC timer, the virtual LAPIC timer has its own set of configuration registers. Although x86 hardware provides APIC virtualization (APICv), APICv only provides a subset of APIC functionality mostly related to interrupt control; there is no such notion as virtual timers in APICv. As typically done when adding a new virtualization hardware capability, we add one bit in the VMX capability register and one in the VM execution control register to enable the guest hypervisor to discover and enable/disable the virtual timer functionality, respectively.

The guest hypervisor can let nested VMs use the virtual timer by setting the bit in the VM execution control register, which is also visible to the host hypervisor. The guest hypervisor sets the virtual timer when first entering the nested VM, either to initialize it after creating the nested VM or to restore the previous timer state when running the nested VM. No further guest hypervisor intervention is needed while the nested VM is running. When the guest hypervisor switches from running a nested VM to running another one, it saves the currently running nested VM state by reading the virtual timer and restores the next nested VM state to the virtual timer.

Virtual timers are designed to be transparent to nested VMs and require no changes to nested VMs. Hardware timers used by nested VMs are transparently remapped by the host hypervisor to virtual timers. When a nested VM programs the hardware timer, it causes an exit to the host hypervisor, which confirms that virtual timers are enabled via the VM execution control register. Rather than forwarding the exit to the respective guest hypervisor to emulate the timer, the host hypervisor handles the exit by programming the virtual timer directly. This can be done either by using software timer functionality or architectural timer support, similar to regular LAPIC timer emulation. Our KVM implementation uses Linux hrtimers to emulate virtual timer functionality. With virtual timers, no guest hypervisor intervention is needed for nested VMs to program timers, avoiding the high cost of existing to the guest hypervisor on frequent programming of the timer by the guest OS in a nested VM.

In emulating the timer, the host hypervisor needs to account for the time difference between the nested VM and the host hypervisor. However, this is already done by existing hypervisors. On x86 systems, a hypervisor keeps the time difference between a VM and itself in a Timestamp Counter (TSC) offset field in the Virtual Machine Control Structure (VMCS). Hardware can access the offset during a VM's execution so that the guest OS can get the correct, current time without a trap. For the same reason, the host hypervisor maintains the time difference between a nested VM and itself in the VMCS for a nested VM. When running a nested VM, the host hypervisor accesses the timer offset the guest hypervisor programmed to a VMCS, combines it with the time difference between itself and the guest hypervisor, and keeps it in the VMCS for a nested VM. Therefore, the host hypervisor can handle the timer operation from a nested VM with the correct offset that it already saved.

Virtual timers provide other timer related operations in a similar way to timer support without DVH. For example, timer interrupts are delivered first from the host hypervisor to the guest hypervisor, which in turn causes timer interrupts to the nested VM. However, unlike regular timers emulated by guest hypervisors, virtual timer support can be further optimized to deliver timer interrupts to the nested VM directly from the host hypervisor using posted interrupts [56]. When the virtual timer for a nested VM is expired, and the host hypervisor gets the control back from the nested VM, the host hypervisor can program hardware through the posted interrupt descriptor to raise an interrupt to the nested VM and return back to the nested VM instead of going back to the guest hypervisor. The only additional information needed for the direct timer interrupt delivery is the interrupt vector number the nested VM programmed for timer interrupts. On the nested VM's programming the interrupt vector number through one of the regular timer configuration registers, the guest hypervisor can pass this information to the host hypervisor via corresponding the virtual timer configuration register.

4.1.3 Virtual IPIs

Guest OSes in VMs send IPIs from one CPU to another. The CPUs controlled by the guest OS are not the physical CPUs, but virtual CPUs which the hypervisor in turn decides when and where to run by scheduling them on physical CPUs. On x86 systems, sending an IPI involves writing the Interrupt Command Register (ICR) with the identifier of the destination CPU. Writing to this register in a VM causes an exit to the hypervisor. The guest OS only knows about virtual CPUs, so the hypervisor determines the physical CPU identifier and does the actual write to the ICR to send the IPI between physical CPUs. Receiving an IPI also causes an exit to the hypervisor, which in turn delivers the IPI to the VM. For nested virtualization, multiple levels of hypervisors must be involved in sending and receiving an IPI. While CPU posted interrupts [56] are available on x86 systems which enable IPIs to be received directly by a VM without exiting to the hypervisor, posted interrupts do not help with the IPI sending side, which still must exit to the guest hypervisor and subsequently through multiple layers until the actual IPI is sent by the host hypervisor.

Figure 4.2 illustrates the seven steps for sending an IPI between virtual CPUs (VCPUs) of an L2 VM, specifically from its VCPU 2 to VCPU 3. Dotted lines indicate what is perceived by each VCPU, while solid lines indicate what actually happens. The guest OS running on the L2 VCPU 2 writes the interrupt number and destination VCPU (VCPU 3) to the ICR and thinks that an IPI is delivered to VCPU 3. Instead, writing to the ICR traps to the L0 hypervisor, which forwards the trap to the L1 hypervisor to emulate the ICR

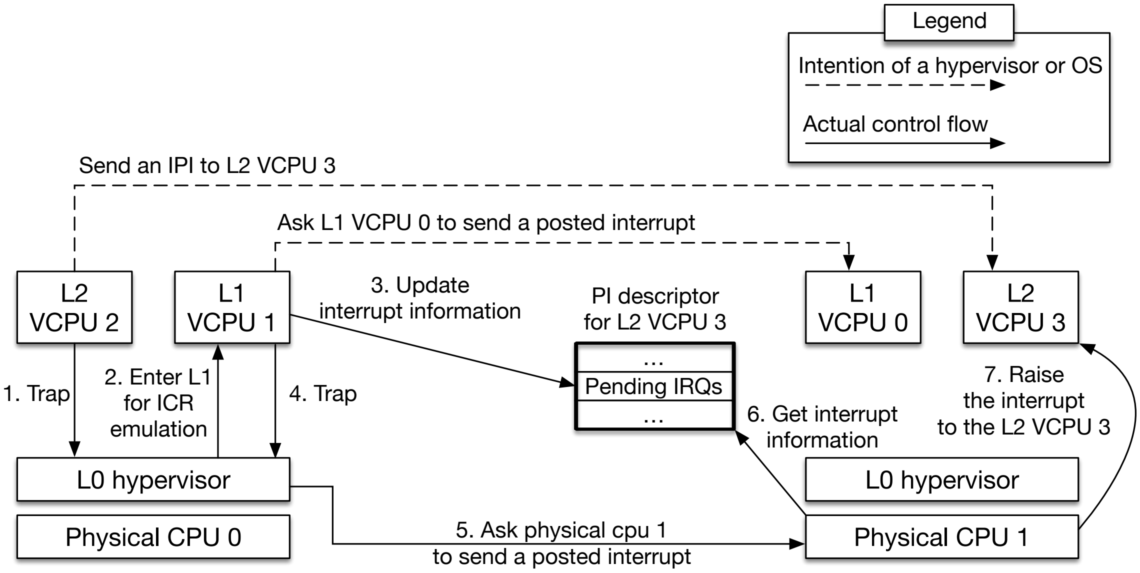

Legend
Intention of a hypervisor or OS
Actual control flow
Send an IPI to L2 VCPU 3
Ask L1 VCPU 0 to send a posted interrupt
L2 VCPU 2
L1 VCPU 1
3. Update interrupt information
PI descriptor for L2 VCPU 3
...
Pending IRQs
...
L1 VCPU 0
L2 VCPU 3
1. Trap
2. Enter L1 for ICR emulation
4. Trap
6. Get interrupt information
7. Raise the interrupt to the L2 VCPU 3
L0 hypervisor
L0 hypervisor
Physical CPU 0
5. Ask physical cpu 1 to send a posted interrupt
Physical CPU 1

Figure 4.2: Nested VM IPI Delivery

behavior. The L1 hypervisor gets the interrupt number and destination VCPU number from the ICR. Assuming that CPU posted interrupts are supported, the L1 hypervisor writes the interrupt number to the posted-interrupt descriptor (PI descriptor) of the destination VCPU. It then asks the L1 VCPU that runs the L2 VCPU 3, the L1 VCPU 0, to raise a posted interrupt to the L2 VCPU 3. This traps to the L0 hypervisor because CPU posted interrupts for the L1 hypervisor are provided by the L0 hypervisor. The L0 hypervisor asks the physical CPU 1 on behalf of the L1 VCPU to raise a posted interrupt. Finally, the physical CPU 1 gets the original IPI information from the PI descriptor and raises an interrupt to the L2 VCPU 3 directly. No hypervisor intervention is necessary on the receiving side, but multiple hypervisors are involved on the sending side.

We introduce virtual IPIs, a DVH technique for reducing the latency of sending IPIs for nested VMs. Virtual IPIs involve two mechanisms, a virtual ICR and a virtual CPU interrupt mapping table. A per virtual CPU virtual ICR is software provided by the host hypervisor that appears to guest hypervisors as an additional hardware capability. We also add one bit in the VMX capability register and one in the VM execution control register to enable the guest hypervisor to discover and enable/disable the virtual IPI functionality, respectively. The guest hypervisor can let nested VMs use virtual IPIs by setting the bit in the VM execution control register, which is also visible to the host hypervisor.

Virtual IPIs are designed to be transparent to nested VMs and require no changes to nested VMs. The hardware ICR used by nested VMs is transparently remapped by the host hypervisor to the virtual ICR. When a nested VM sends an IPI by writing the ICR, it causes an exit to the host hypervisor, which confirms that virtual IPIs are enabled via the VM execution control register. Rather than forwarding the exit to the respective guest hypervisor, the host hypervisor handles the exit by emulating the IPI send operation and writing the hardware ICR directly. Using virtual IPIs, no guest hypervisor intervention is needed for nested VMs to send IPIs.

To send the IPI, the host hypervisor must know the destination physical CPU that runs the IPI destination virtual CPU of the nested VM. A hypervisor, however, typically only knows how virtual CPUs of its own VMs are distributed on physical CPUs; it does not know the information for nested VMs. Unlike virtual-passthrough and virtual timers, the host hypervisor cannot get the nested VM virtual CPU distribution information through existing hardware interfaces provided to the guest hypervisor.

To address this problem, we add new virtual hardware interfaces for guest hypervisors, the virtual CPU interrupt mapping table and the virtual CPU interrupt mapping table address register (VCIMTAR). This table is a per VM global structure in memory that provides mappings from virtual CPUs to the physical CPUs maintained by the guest hypervisors. The guest hypervisor can share the mapping information with the host hypervisor by programming the table's base memory address to the VCIMTAR, which enables the host hypervisor to find the destination physical CPU running the IPI destination nested VM's virtual CPU. On x86, each table entry has a mapping from virtual CPU number to the corresponding PI descriptor, which includes a physical CPU number, to fully leverage posted interrupts for nested VMs on the receiving side.

Figure 4.3 shows the same nested VM IPI delivery example from Figure 4.2, but using virtual IPIs. The guest OS running on the L2 VCPU 2 writes to the ICR as before, but the trap is handled by the L0 hypervisor directly with virtual IPIs. The L1 hypervisor is not involved. The L0 hypervisor gets the interrupt number and destination VCPU number from the ICR. However, it does not know the location of the PI descriptor for the destination L2 VCPU; it can only access the PI descriptor of the currently running VCPU on the current physical CPU, the L2 VCPU 2 in this example. With virtual IPIs, the L0 hypervisor looks up the correct destination PI descriptor in the virtual CPU interrupt mapping table using the destination VCPU number (L2 VCPU 3) as the key. It then can update the PI descriptor in the same way as the L1 hypervisor would do, then asks the physical CPU 1 to raise a posted interrupt. Finally, the physical CPU 1 gets the original IPI information from the PI descriptor and raises an interrupt to the L2 VCPU 3 directly. No hypervisor intervention is necessary on the receiving side, and only host hypervisor intervention is needed on the sending side.

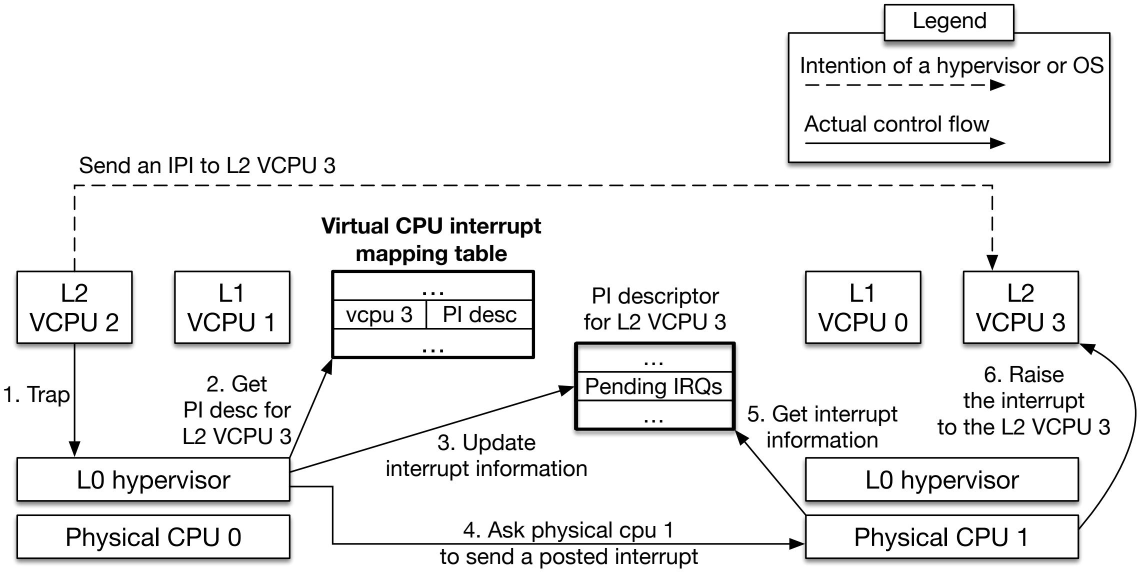

Legend
Intention of a hypervisor or OS
Actual control flow
Send an IPI to L2 VCPU 3
Virtual CPU interrupt mapping table
...
vcpu 3
PI desc
...
L2 VCPU 2
L1 VCPU 1
PI descriptor for L2 VCPU 3
...
Pending IRQs
...
L1 VCPU 0
L2 VCPU 3
1. Trap
2. Get PI desc for L2 VCPU 3
3. Update interrupt information
5. Get interrupt information
6. Raise the interrupt to the L2 VCPU 3
L0 hypervisor
L0 hypervisor
Physical CPU 0
4. Ask physical cpu 1 to send a posted interrupt
Physical CPU 1

Figure 4.3: Nested VM IPI Delivery with Virtual IPIs

4.1.4 Virtual Idle

OSes execute idle instructions, such as the HLT (halt) instruction on x86, to enter CPU low-power mode when possible. When an idle instruction is executed in a VM, the hypervisor will typically trap the instruction to retain control of the physical CPU. The hypervisor then can switch to other tasks of its own or enter the real low-power mode if it does not have jobs to run. The hypervisor will return to the VM later when the VM receives new events to handle. For nested virtualization, multiple levels of hypervisors are involved in entering and exiting low-power mode, resulting in increased interrupt delivery latencies for nested VMs.

We introduce virtual idle, a DVH technique for reducing the latency of switching to and from low-power mode in nested VMs. Virtual idle leverages existing architectural support for configuring whether to trap the idle instruction, but uses it in a new way. We configure the host hypervisor to trap the idle instruction as before, but all guest hypervisors to not trap it. The host hypervisor knows not to forward the idle instruction trap to the guest hypervisor since it can access the guest hypervisor's configuration for nested VMs through the VMCS as discussed for virtual timers in Section 4.1.2. A nested VM executing the idle instruction will only trap to the host hypervisor, and the host hypervisor will return to the nested VM directly on a new event. As a result, the cost of switching to and from low-power mode for nested VMs using virtual idle will be similar to that for non-nested VMs, avoiding guest hypervisor interventions.

Currently available options such as disabling traps [71] in all hypervisors or using a guest kernel option to poll [103] instead of executing the idle instruction can also reduce latency similar to virtual idle. The key difference is that those options simply consume and waste physical CPU cycles when the nested VM does nothing. With virtual idle, the host hypervisor only runs the nested VM when it has jobs to run.

Virtual idle can be used whenever desired by a guest hypervisor. However, instead of enabling virtual idle all the time when running a nested VM, we enable it only when the guest hypervisor knows it has no other nested VMs that it can run. When there is nothing else to run if the running virtual CPU of the nested VM goes idle, it is best to allow the host hypervisor to handle the idle instruction since returning to the guest hypervisor has no benefit. However, when there are other nested VMs that can be run by the guest hypervisor, it is useful to return to the guest hypervisor to allow it to schedule another nested VM to execute. Otherwise, the host hypervisor will schedule the CPU to run other VMs that it knows about and may not include any other nested VMs managed by the respective guest hypervisor because it thinks the idle instruction execution indicates that the guest hypervisor has no other jobs to run.

4.1.5 Recursive DVH

DVH can be easily used with additional levels of nested virtualization. Guest hypervisors that used to use virtual hardware transparently for its VMs for two levels of virtualization now need to expose the virtual hardware to the next level guest hypervisors recursively. Only the last level guest hypervisor uses virtual hardware for its VM transparently as before. Once guest hypervisors at any level *k* provide virtual hardware to the next level, the guest hypervisors get information from the next level guest hypervisors at level *k+1*, translate the information valid at level *k*, and program the information to virtual hardware provided so that hypervisors at level *k-1* can access the information in turn. In that way, the host hypervisor will have all the necessary information to emulate nested VMs. The currently running guest OS in a nested VM can always use the virtual hardware without trapping to guest hypervisors.

For example, recursive virtual timers can be achieved with the support from the guest hypervisors. Each guest hypervisor except the last one provides a virtual timer, including bits in the VMX capability and VM execution control registers, to the next level hypervisor. The last level hypervisor, which is equivalent to the guest hypervisor for two levels of virtualization, does not provide a virtual timer for its VM, but transparently allows it to use the virtual timer provided to the last level hypervisor. The last level hypervisor can decide whether to enable or disable the virtual timer feature for its VM, but all other guest hypervisors will only enable the virtual timer for its nested VMs if its respective next level hypervisor enables it. For example, the L1 hypervisor will only enable virtual timers for an L3 VM if both the L1 and the L2 hypervisors enable it for their respective VMs. In this way, the enable bits of all guest hypervisors are combined using an *and* operation into the single enable bit that the L1 hypervisor sets for an L_n VM. The L0 hypervisor would use the virtual timer for the L_n VM if the L1 hypervisor enabled the virtual timer, which means all other guest hypervisors also enabled it. If the L1 hypervisor disabled the virtual timer, then the L_k hypervisor will forward the L_n VM timer access to the L_{k+1} hypervisor recursively, where k starts from 0, until a hypervisor L_i finds a hypervisor L_{i+1} with the enable bit set or control reaches to the $L_{n\text{-}1}$ hypervisor. For both cases, the respective hypervisor emulates timer functionality for the L_n VM.

4.1.6 DVH Migration

Because DVH provides virtual hardware, including virtual I/O devices, in software, it allows the host hypervisor to encapsulate the state of the L1 VM and decouple it from physical devices to support migration. From the perspective of the host hypervisor, migrating an L1 VM that contains or does not contain a nested VM is essentially the same. The nested VM using DVH does not introduce additional hardware dependencies on the host and is completely encapsulated by the host hypervisor. For example, a hypervisor supporting migration of VMs that use virtual I/O devices naturally supports migration of VMs that use virtual-passthrough.

The only difference from the perspective of the host hypervisor between a VM with and without DVH is that the former provides more virtual hardware to a VM, such as a virtual IOMMU and virtual timer, while the latter does not. Migration using DVH requires that the state associated with the additional virtual hardware is also migrated. This is no different than migrating any VM using any other virtual hardware in which the hardware state must be properly saved and restored. DVH is software only and is not coupled to any physical device, making it straightforward for the hypervisor to encapsulate its state for migration.

When migrating a nested VM, without its L1 VM, the level of virtual hardware support required depends on the DVH technique. For all of the DVH techniques discussed other than virtual-passthrough, the level of support needed is minimal. Virtual timers, virtual IPIs, and virtual idle do not introduce any additional virtual hardware state that needs to be migrated compared to what would be required if the guest hypervisor itself were emulating that state without DVH. For virtual IPIs and virtual idle, the techniques are stateless, and there is no additional state that needs to be saved for nested VM migration. For virtual timers, the guest hypervisor needs to save the timer value for nested VM migration, just as it would if it were handling timer emulation itself without DVH. This simply involves getting the timer value from the virtual hardware instead of from the guest hypervisor's emulated hardware. The timer offset also needs to be saved, but that is already saved as part of the VM state stored in VMCS, with or without DVH. For virtual-passthrough, we have already discussed migrating a nested VM in Section 3.2.4.

4.2 Evaluation

We implemented the four DVH mechanisms in KVM and evaluated their performance. Experiments used x86 server hardware in CloudLab [41], each with two Intel Xeon Silver 4114 10-core 2.2 GHz CPUs (hyperthreading disabled), 192 GB ECC DDR4-2666 RAM, an Intel DC S3500 480 GB 6G SATA SSD, and a dual-port Intel X520-DA2 10Gb NIC

(PCIe v3.0, 8 lanes). The servers include VMCS Shadowing [55] for nested virtualization, APICv for virtual interrupt support and posted interrupts from CPUs, and VT-d IOMMU support for direct device assignment with posted interrupt support from devices.

To provide comparable measurements, we kept the software environments the same as much as possible. All hosts and VMs used Ubuntu 14.04 with the same Linux 4.18 kernel and software configuration unless otherwise indicated. We fixed a KVM hypervisor bug related to using virtualization support for accessing segment registers, which has since been incorporated into later versions of KVM [25]; all our measurements included this fix for a fair comparison. For the host and guest hypervisors, we used KVM with QEMU 3.1.0. When using virtual I/O devices with KVM, with or without virtual-passthrough, we used the standard virtio network device with vhost-net and the `cache=none` setting for virtual block storage devices [66, 99, 51].

We use newer x86 servers having posted interrupt support from devices to fully leverage virtualization hardware support, which is missing in servers we used in Chapter 2 and Chapter 3. We also use up-to-date KVM and QEMU versions to keep up with upstream changes and improvements.

We used four different configurations for our measurements: (1) native: running natively on Linux with 4 cores and 12 GB RAM, (2) VM: running in a VM with 4 cores and 12 GB RAM on a hypervisor with 6 cores and 24 GB RAM, (3) nested VM: running in an L2 VM with 4 cores and 12 GB RAM on an L1 hypervisor with 6 cores with 24 GB RAM on an L0 hypervisor with 8 cores and 36 GB RAM, (4) L3 VM: running in an L3 VM with 4 cores and 12 GB RAM on an L2 hypervisor with 6 cores with 24 GB RAM on an L1 hypervisor with 8 cores and 36 GB RAM on an L0 hypervisor with 10 cores and

Name	Description
Hypercall	Switch from VM to hypervisor and immediately back to VM without doing any work in the hypervisor.
DevNotify	Device notification via MMIO write from VM virtio device driver to virtual I/O device.
ProgramTimer	Program LAPIC timer in TSC-Deadline mode.
SendIPI	Send IPI to CPU that is idle which needs to wakeup and switch to running destination VM vCPU to receive IPI.

Table 4.1: Virtualization Microbenchmarks

48 GB RAM. Two cores and 12 GB RAM were added for the hypervisor at each virtualization level similar to previous work [109, 105] on nested virtualization using multicore processors. We pinned each virtual CPU to a specific physical CPU following the best measurement practices [30, 96, 117]. For benchmarks that involve clients interacting with the server, the server ran on the configuration being measured while the clients ran on a separate dedicated machine, ensuring that clients were never saturated during our experiments. Clients ran natively on Linux with the same kernel version as the server and were configured to use the full hardware available.

We evaluated performance using microbenchmarks and widely-used application workloads, as listed in Table 4.1 and Table 4.2, respectively. Other than DVH, no changes were required to the hypervisors except the KVM bugfix, which was used for all configurations. DVH required changes in the hypervisors to provide and use the virtual hardware. We also implemented posted interrupt support in the virtual IOMMU for DVH measurements, which is missing in QEMU, to fully leverage the benefits of the DVH design.

Table 4.3 shows performance measurements from running the microbenchmarks in a VM, nested VM, nested VM using DVH, L3 VM, and L3 VM using DVH. Additional virtualization levels are not supported by KVM [59]. Measurements were run using paravirtual I/O, though only DevNotify uses the I/O device. The measurements show more

Name	Description
Netperf	netperf v2.6.0 [60] server running with default parameters on the client in three modes: TCP_RR, TCP_STREAM, and TCP_MAERTS, measuring latency and throughput, respectively.
Apache	Apache v2.4.7 Web server running ApacheBench [100] v2.3 on the remote client, measuring requests handled per second serving the 41 KB file of the GCC 4.4 manual using 10 concurrent requests.
Memcached	memcached v1.4.14 using the memtier benchmark v1.2.3 with its default parameters.
MySQL	MySQL v14.14 (distrib 5.5.41) running SysBench v.0.4.12 using the default configuration with 200 parallel transactions.
Hackbench	hackbench [93] using Unix domain sockets and 100 process groups running with 500 loops.

Table 4.2: Application Benchmarks

than an order of magnitude increase in cost when running in a nested VM versus a VM. Hypercall is much more expensive in a nested VM than in a VM as it takes much longer to exit to the guest hypervisor from a nested VM than to exit from a VM to its hypervisor without nested virtualization. As expected, DVH does not improve nested VM performance for Hypercall as it always requires exiting to the guest hypervisor.

DVH substantially improves nested VM performance for the other microbenchmarks as each of them exercises one of the DVH mechanisms to avoid exits to the guest hypervisor. Compared to vanilla KVM running the nested VM, DVH provides more than 3 times better performance on DevNotify due to virtual-passthrough, 13 times better performance on ProgramTimer due to virtual timers, and 8 times better performance on SendIPI due to virtual IPI and virtual idle. SendIPI measures the total time to send and receive an IPI when the VM is idle on the destination CPU.

Although DVH performs much better than vanilla KVM in all cases, it incurs noticeably more overhead running a nested VM than running a VM for DevNotify. The extra cost is a result of the host hypervisor needing to walk the extended page table (EPT) of the VM to check if a fault occurred because the mapping does not exist at the faulting address in

	VM	**nested VM**	**nested VM + DVH**	**L3 VM**	**L3 VM + DVH**
Hypercall	1,575	37,733	38,743	857,578	929,724
DevNotify	4,984	48,390	13,815	1,008,935	15,150
ProgramTimer	2,005	43,359	3,247	1,033,946	3,304
SendIPI	3,273	39,456	5,116	787,971	5,228

Table 4.3: Microbenchmark Performance in CPU Cycles

the EPT. Once the host hypervisor confirms that the mapping is valid, it handles the fault directly. Note that no data is transferred in this microbenchmark and more realistic I/O device usage that accesses data would have much less overhead for running a nested VM with DVH compared to just running a VM.

L3 VM measurements show more than a 200 times increase in cost compared to VM due to excessive exit multiplication with further virtualization levels. DVH again substantially improves L3 VM performance for all microbenchmarks other than Hypercall, more than 150 times on average. More importantly, using DVH resulted in similar performance for both L3 and L2 VMs, an expected outcome since DVH removes guest hypervisor interventions. Our results show how DVH significantly improves nested virtualization performance. By resolving the exit multiplication problem, DVH achieves performance close to non-nested virtualization performance regardless of the nested virtualization level.

Figure 4.4 shows performance measurements from running the application workloads in six different VM configurations. We considered all possible network I/O configurations. For VM, we measured both paravirtual I/O and passthrough. For nested VM, we measured paravirtual I/O, passthrough, DVH, and DVH with only the virtual-passthrough mechanism enabled, denoted as DVH-VP, to provide a conservative comparison against passthrough. DVH-VP did not require any hypervisor changes to support virtual hardware; it did not

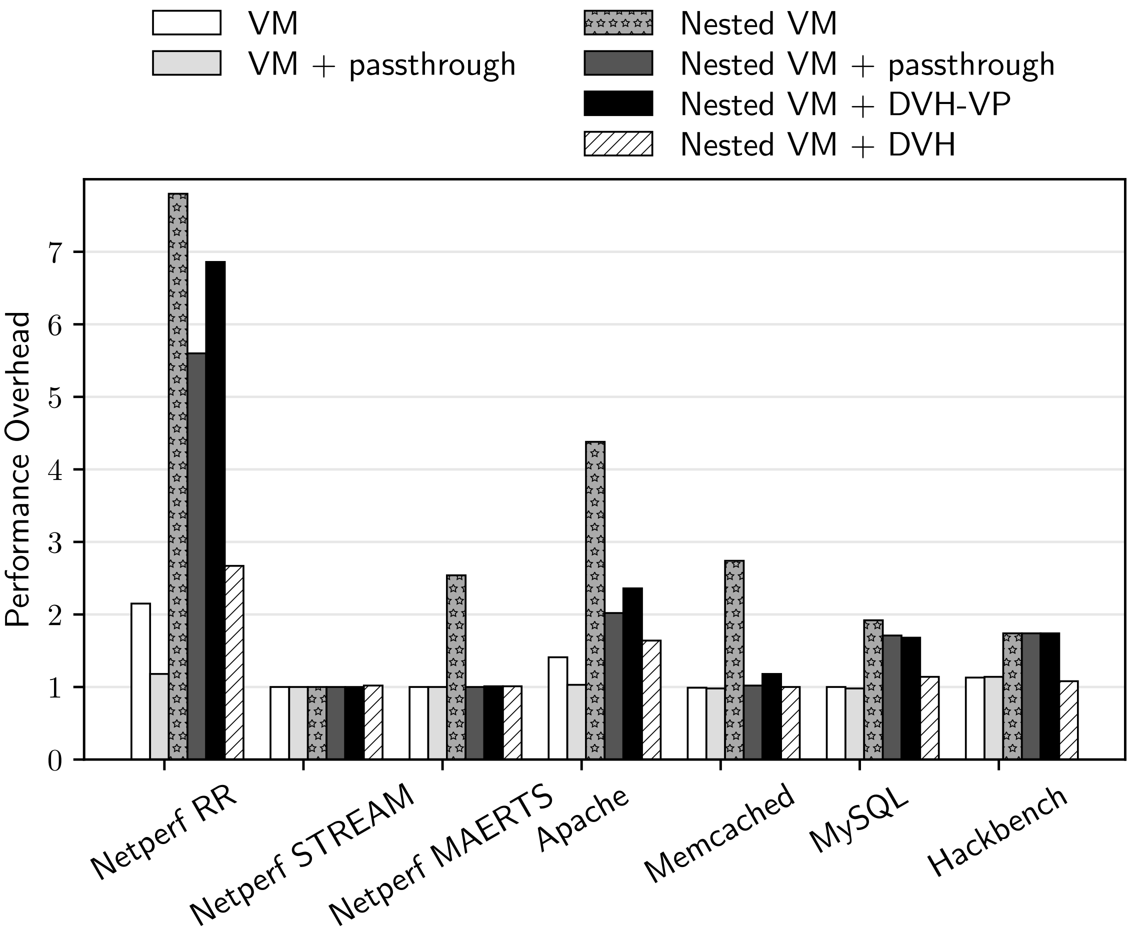


Figure 4.4: Application Performance

include posted interrupt support in the virtual IOMMU. Since we are more interested in overhead than absolute performance, VM and nested VM performance are normalized relative to native execution, with lower meaning less overhead. Table 4.4 shows the non-normalized results from the measurements. Note that the numbers are slightly different from Table 3.3 because of differences in hardware and the Linux kernel/QEMU versions used.

For the VM case, both paravirtual I/O and passthrough provide mostly similar performance, with passthrough having better performance for both Netperf RR and Apache.

Application	Unit	Baseline	L1	L2	L2 + DVH	L3	L3 + DVH
Netperf RR	trans/sec	45,578	21,208	5,843	17,058	361	15,392
Netperf Stream	Mbits/sec	9,413	9,414	9,411	9,229	3,969	9,162
Netperf Maerts	Mbits/sec	9,414	9,412	3,700	9,329	294	9,273
Apache	trans/sec	15,469	10,940	3,534	9,409	157	7,720
Memcached	trans/sec	354,132	358,547	129,118	352,606	3,282	293,087
MySQL	sec	4.4	4.4	8.5	5.1	96.2	7.2
Hackbench	sec	10.4	11.8	18.0	11.2	143.7	14.1

Table 4.4: Application Benchmark Raw Performance

The virtual I/O device model overall provides sufficient performance for the VM case with passthrough providing only marginal gains for most of the application workloads. Since Hackbench does not use I/O, it shows no performance difference between different I/O models.

For the nested VM case, performance differences among the different VM configurations are substantial. Only DVH is able to provide nested virtualization performance almost as good as the VM case for all application workloads. DVH performance can be more than 3 times better than just using paravirtual I/O, and more than 2 times better than passthrough. While paravirtual I/O performs much worse than passthrough for most application workloads, more than 3 times worse than the VM case for Apache, Memcached, Netperf RR, and Netperf MAERTS, DVH-VP alone delivers nested VM performance comparable to passthrough for most application workloads. Performance gains using DVH-VP instead of the virtual I/O device model are substantial, more than doubling performance for Apache and almost tripling performance for Memcached. Note that the virtual I/O device emulation done by the host hypervisor using DVH-VP is almost identical to that using virtual I/O model; it relays data between the physical I/O device and (nested) VM address space. The performance gain using DVH-VP is a result of removing the guest hypervisor's

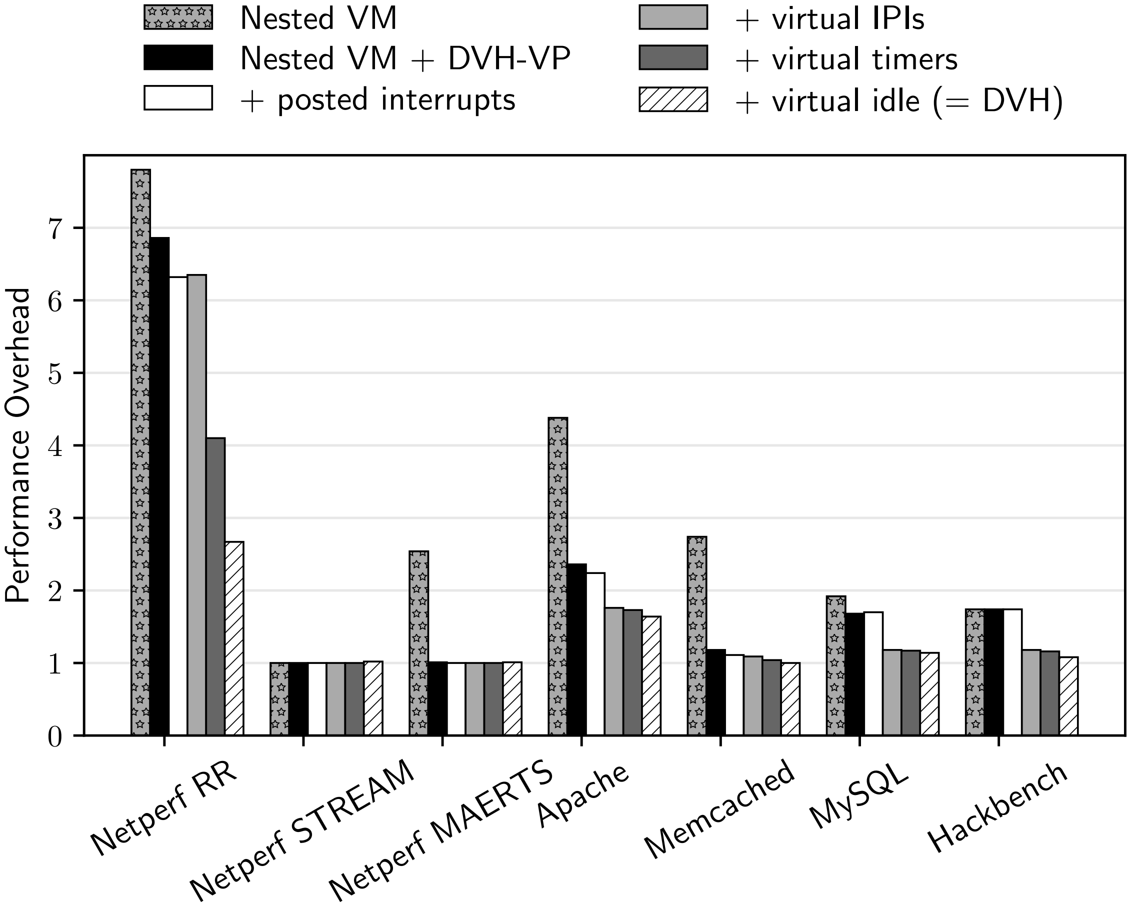


Figure 4.5: Application Performance Breakdown

intervention on physical CPUs that run the nested VM.

Figure 4.5 provides a finer granularity breakdown of the nested virtualization performance in Figure 4.4 to show how incrementally applying each DVH technique affects performance. Starting with DVH-VP, we show how performance changes by adding posted interrupt support in the virtual IOMMU, virtual IPIs, virtual timers, and virtual idle, respectively, the latter including all DVH techniques. Different DVH techniques improve performance to varying degrees for different application workloads. Virtual IPIs most improve performance for Apache, MySQL, and Hackbench. Virtual timers improve performance most for Netperf RR, and help some with Apache and MySQL. Virtual idle improves performance for Netperf RR as the workload often goes idle. The different DVH techniques also have performance interactions. For example, for Memcached, each of the individual DVH techniques except virtual idle improves performance significantly when used by itself, but once one technique is used, the other techniques do not help much further because there is not much overhead left. On the other hand, virtual idle helps significantly with Netperf RR, but only when used in combination with the other DVH techniques, not by itself.

Figure 4.6 shows measurements using three levels of virtualization. Only DVH is able to provide nested virtualization performance almost as good as the VM case for all application workloads. DVH performance is up to two orders of magnitude better than just using paravirtual I/O and can be more than 30 times better than passthrough. In contrast, these measurements show that adding an additional level of virtualization makes paravirtual I/O performance practically unusable, showing more than two orders of magnitude overhead for multiple workloads such as Memcached and Apache, and much worse than the passthrough model. DVH-VP alone again continues to offer similar performance as passthrough, though it still performs multiple times worse than native execution and not as well as DVH. Table 4.4 shows the non-normalized results from the measurements.

Figure 4.7 and Figure 4.8 show total migration time for VM and nested VM, respectively, in seconds for running the same workloads on x86 as used in Figure 4.4. To provide a baseline for comparison, we also measured the time to migrate VM and nested VM when not running any workload, denoted as Idle. Live migration was done between two identical x86 servers on the same subnet. We set the transfer bandwidth to 32 Gbps, which

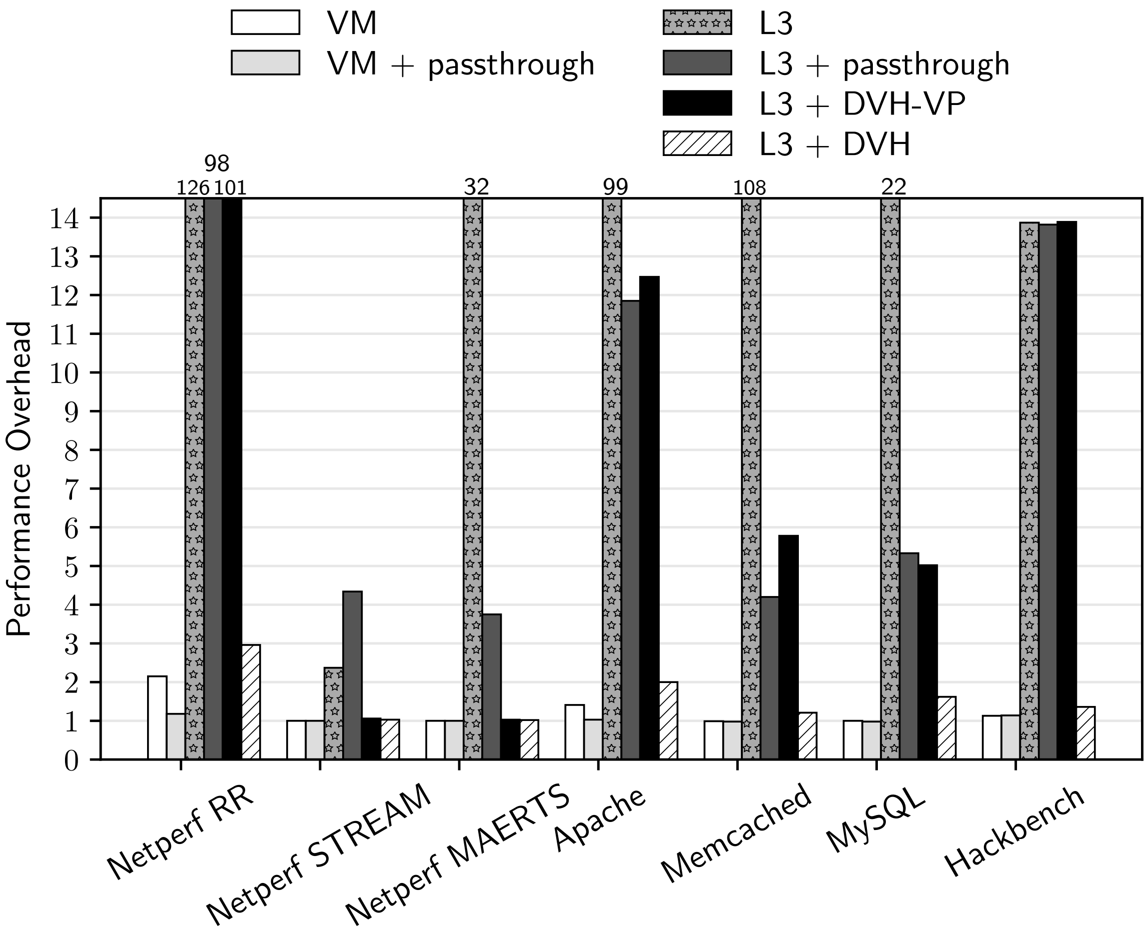


Figure 4.6: Application Performance in L3 VM

effectively imposes no limit on the bandwidth available for migration. We used a different configuration from the default transfer bandwidth used in Chapter 3 for migration measurements because Hackbench, which we did not use in Chapter 3, could not be migrated with the default transfer bandwidth because it generated dirty pages faster than the default transfer bandwidth.

Figure 4.7 shows that paravirtual I/O, DVH-VP, and DVH have similar migration performance in terms of total migration time when migrating a VM running a nested VM in it. In contrast, none of the passthrough configurations provide any migration capability,

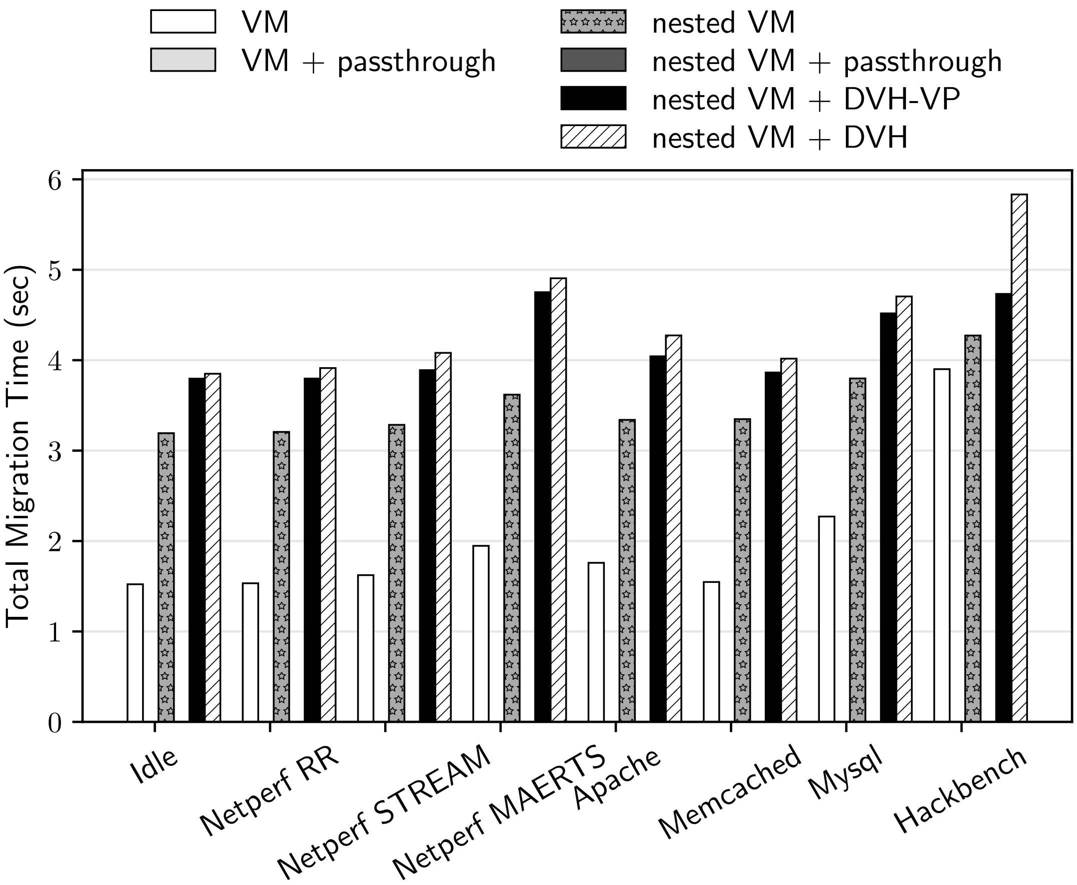


Figure 4.7: Total VM Migration Time on x86

as represented by the blank measurements for those configurations. In general, DVH-VP and DVH take more time than paravirtual I/O due to QEMU taking longer to identify zero pages for DVH-VP and DVH compared to paravirtual I/O. For example in the Idle case, the migration time for DVH was about .6 s longer than for paravirtual I/O and almost all of that difference was just due to the extra time to identify zero pages.

DVH takes slightly more time than DVH-VP since DVH has more states to migrate because of additional virtual hardware. Although total migration time shows modest differences between paravirtual I/O, DVH-VP, and DVH, the actual downtime took less than 300 ms for all configurations, showing that migration can be done with very little impact on application availability.

Note that migration with the VM configuration is faster than with the nested VM configuration since the latter has substantially more state to migrate because we added more memory and CPU for each virtualization level. Running the workload in the nested VM on top of the VM also results in more dirty pages to migrate.

Figure 4.8 shows the VM migration times for just migrating the nested VM without migrating the underlying VM. For comparison, Figure 4.8 also shows the VM migration time for the non-nested VM configuration from Figure 4.7. In contrast to Figure 4.7, the time for migration with the nested VM configuration is much closer to that of the VM configuration since migrating the nested VM no longer involves also migrating the underlying VM, resulting in migrating a similar amount of state as non-nested VM migration. Just like Figure 4.7, none of the passthrough configurations provide any migration capability, as represented by the blank measurements for those configurations. DVH-VP and DVH show increased migration time compared to paravirtual I/O due to the same zero page issue. For example in the Idle case, the migration time for DVH was about .7 s longer than for paravirtual I/O and almost all of that difference was just due to the extra time to identify zero pages. Note that while Figure 4.8 shows that percentage increase in migration time for DVH and DVH-VP versus paravirtual I/O is higher than in Figure 4.7, this is because the total migration time in Figure 4.8 for nested VM configurations is much less than in Figure 4.7. As a result, the overhead due to zero pages accounts for a relatively larger percentage of the migration time in Figure 4.8 even though the actual time difference for DVH and DVH-VP versus paravirtual I/O is similar to Figure 4.7.

In contrast to Figure 4.7, DVH-VP and DVH show almost the same migration time for

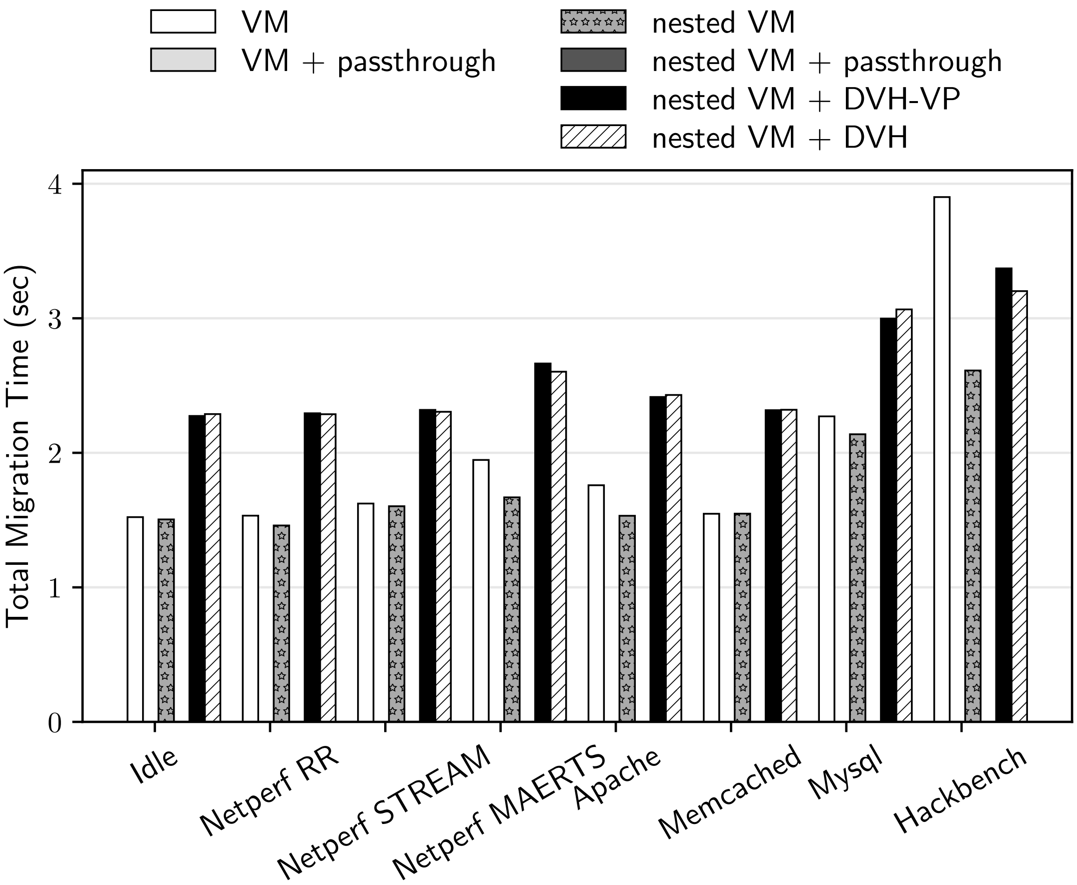


Figure 4.8: Total Nested VM Migration Time on x86

each workload in Figure 4.8. The additional virtual hardware DVH introduced other than virtual I/O devices is not part of the nested VM state, resulting in DVH-VP and DVH having a a similar amount of state to migrate. Like the non-nested VM migration results, the downtime for migrating the nested VM took less than 300 ms for paravirtual I/O, DVH-VP, and DVH. These results show that DVH provides a unique combination of superior performance for nested virtualization while supporting migration, a key virtualization feature for cloud computing deployments.

Although the paravirtual I/O nested VM configuration has the fastest migration times in Figure 4.8 for many workloads, this is in part due to the nested VM configuration having the worst performance as shown in Figure 4.4. It has the worst performance in part because the workloads experience longer idle periods, resulting in the CPU being less busy running the workloads and having more time available to perform the migration itself, which reduces migration time. It also has the worst performance in part because the workloads run slower, resulting in less data being written to memory and therefore fewer dirty pages that need to be migrated, also reducing migration time. For example for Hackbench, the nested VM migration time is much faster than the non-nested VM because it runs much slower and so does not generate dirty pages as fast, resulting in less state to migrate.

4.3 Related Work

Modern architectures such as x86 and Arm have been adding more powerful virtualization extensions to enhance VM and nested VM performance [56, 55, 9, 12, 70, 31, 33, 34, 24]. Hardware extensions such as APICv on x86 [56] and VGIC on Arm [9, 12] provide additional hardware state that can be dedicated for use by VMs and nested VMs. DVH provides additional virtual hardware, but as a software solution that does not require additional hardware. DVH can be deployed in addition to and in the absence of hardware extensions to improve nested virtualization performance. It can also be used to evaluate future hardware extensions. Hardware extensions specific to nested virtualization such as VMCS shadowing on x86 [55] and NEVE on Arm, as introduced in Chapter 2, reduce the cost of guest hypervisor execution, but they do not avoid guest hypervisor interventions for nested VMs. In contrast, DVH removes multiple levels of guest hypervisor interventions and replaces them with much less expensive host hypervisor interventions. DVH and architectural support for nested virtualization are complementary; DVH works on top of the hardware extensions, as shown in Section 4.2.

Denali [108] proposed a different virtual interface from the underlying hardware to VMs, provided by the software running directly on the hardware to improve virtualization scalability. Fluke [44] provided a different interface to VMs to support OS extensibility. These approaches do not support legacy OSes and hypervisors. In contrast, DVH shows how virtual hardware can be provided directly through multiple layers of hypervisors to improve nested virtualization performance, in a way that is transparent and does not require changes to the nested VMs.

Dichotomy [109] proposed migrating nested VMs from the guest hypervisor to the host hypervisor to reduce the overhead of nested virtualization, then migrating them back when guest hypervisor intervention is required. While this approach provides marginal performance gain, virtual I/O migration across different hypervisors would require significant implementation or even not be possible. Virtual-passthrough provides virtual I/O devices in the host hypervisor to nested VMs directly without migration, enabling it the work regardless of virtual I/O device types guest hypervisors support.

DID [103] proposed an x86 mechanism to allow VMs to program physical timers without trapping for single-level virtualization by restricting hypervisors to use a timer on a designated core. This mechanism is based on their design that all interrupts are delivered to the VM natively but does not fully leverage posted-interrupt hardware support commonly used by x86 hypervisors. DVH, in contrast, takes a different approach to provide an additional timer for VMs and is designed to work on hypervisors leveraging modern architectural support for virtualization.

4.4 Summary

We introduced DVH, a new approach for directly providing virtual hardware to nested virtual machines without the intervention of multiple levels of hypervisors, extending the idea of virtual-passthrough. Beyond virtual-passthrough, we introduce three additional DVH mechanisms, virtual timers to transparently remap timers used by nested VMs to virtual timers provided by the host hypervisor, virtual inter-processor interrupts that can be sent and received directly from one nested virtual processor to another, and virtual idle that enables nested VMs to switch to and from low-power mode without guest hypervisor interventions. DVH provides virtual hardware for these mechanisms that mimics the underlying hardware and, in some cases, adds new enhancements that leverage the flexibility of software without the need for matching physical hardware support. Like virtual-passthrough, DVH reduces exit multiplication by eliminating the need for guest hypervisor execution when the nested VM accesses the virtual hardware. We have implemented DVH in KVM. Our experimental results show that combining the four DVH mechanisms can provide even greater performance than virtual-passthrough alone and provide near-native execution speeds on real application workloads even for multiple levels of recursive virtualization. We also show that DVH can provide better performance than device passthrough while at the same time enabling migration of nested VMs, thereby providing a combination of both good performance and key virtualization features not possible with device passthrough.

Chapter 5

Conclusions and Future Work

This dissertation explored new approaches to enhance nested virtualization performance and showed that simple changes to hardware, software, and virtual machine configuration that are transparent to nested virtual machines can provide near-native execution speed for real application workloads.

First, we presented the first in-depth study of Arm nested virtualization. We introduce a novel paravirtualization technique to evaluate the performance of new architectural features before hardware is readily available. Using this technique, we evaluate Armv8.3 nested virtualization support and find that its performance is prohibitively expensive compared to normal virtualization, despite its similarities to x86 nested virtualization. We show how differences between Arm and x86 in non-nested virtualization support end up causing significant exit multiplication on Arm. To address this problem, we introduce NEVE, simple architecture extensions that provide register redirection, and coalesce and defer traps by logging system register accesses from the guest hypervisor to memory and only copying the results of those accesses to hardware system registers when necessary. This reduces exit multiplication by batching the handling of multiple hypervisor instructions on one exit instead of exiting for each individual hypervisor instruction executed by the guest hypervisor. We evaluate the performance of NEVE and show that NEVE can improve nested virtualization performance by an order of magnitude on real application workloads compared to the Armv8.3 architecture, and can provide up to three times less virtualization overhead than x86. NEVE is straightforward to implement in Arm and is included the Armv8.4 architecture.

Second, we presented virtual-passthrough, a novel yet simple technique for boosting I/O performance when using nested virtualization. Virtual-passthrough is similar to direct physical device assignment but instead assigns virtual I/O devices to nested VMs. Virtual devices provided by the host hypervisor can be assigned to nested VMs directly without delivering data and control through multiple layers of virtual I/O devices. Therefore, virtual-passthrough reduces exit multiplication by eliminating the need for guest hypervisor execution when the nested VM interacts with the assigned virtual I/O devices. The approach leverages the existing direct device assignment mechanism and implementation, so it only requires virtual machine configuration changes. Virtual-passthrough preserves I/O interposition in the host hypervisor different from physical device passthrough while virtual-passthrough also can easily support important I/O interposition benefits such as migration in the hypervisors at intermediate layers. Scalability is not a problem as many virtual devices can be supported by a single physical device. Supporting both paravirtual and emulated I/O devices is straightforward. The technique is platform agnostic, does not require hardware support such as physical IOMMUs or SR-IOV. We have applied virtual-passthrough in KVM for both x86 and Arm hardware, and show that it can provide more than an order of magnitude improvement in performance over current KVM virtual device support on real application workloads.

Third, we introduced DVH, a new approach for directly providing virtual hardware to nested virtual machines without the intervention of multiple levels of hypervisors, extending the idea of virtual-passthrough. Beyond virtual-passthrough, we introduce three additional DVH mechanisms, virtual timers to transparently remap timers used by nested VMs to virtual timers provided by the host hypervisor, virtual inter-processor interrupts that can be sent and received directly from one nested virtual processor to another, and virtual idle that enables nested VMs to switch to and from low-power mode without guest hypervisor interventions. DVH provides virtual hardware for these mechanisms that mimics the underlying hardware and, in some cases, adds new enhancements that leverage the flexibility of software without the need for matching physical hardware support. Like virtual-passthrough, DVH reduces exit multiplication by eliminating the need for guest hypervisor execution when the nested VM accesses the virtual hardware. We have implemented DVH in KVM. Our experimental results show that combining the four DVH mechanisms can provide even greater performance than virtual-passthrough alone and provide near-native execution speeds on real application workloads. We also show that DVH can provide better performance than device passthrough while at the same time enabling migration of nested VMs, thereby providing a combination of both good performance and key virtualization features not possible with device passthrough.

In general, there are three different optimization points to reduce exit multiplication: 1) to reduce the cost of individual exit to the host hypervisor, 2) to reduce the number of exits from the guest hypervisor to the host hypervisor, 3) to reduce the number of exits from the nested VM to the guest hypervisor. The performance gain of optimizing the first point is limited since the cost of an individual exit is important for non-nested virtualization as well and is already highly optimized in software and hardware. The performance gain of optimizing the third point can deliver the highest benefit since eliminating exits to the guest hypervisor avoids running the guest hypervisor and therefore also eliminates exits from the guest to host hypervisor.

This dissertation presents mechanisms optimizing the second and the third points to reduce the number of exits unique for nested virtualization. The mechanisms involve simple changes to hardware, software, and VM configuration. NEVE introduces simple hardware changes to reduce the number of exits from the guest to host hypervisor; it supports running completely unmodified guest hypervisors and OS software. Virtual-passthrough introduces simple VM configuration changes to reduce the number of exits to the guest hypervisor for I/O operations executed by the nested VM; it requires no hardware or software changes. DVH introduces simple software changes to reduce the number of exits to the guest hypervisor for a broader range of instructions executed by the nested VM; it requires software changes in the host hypervisor to provide virtual hardware and in the guest hypervisor to use the virtual hardware. This dissertation has shown that the three mechanisms are transparent to nested VMs and can provide near-native execution speed for real application workloads.

This dissertation also introduces powerful new ways of thinking about and using virtual hardware. For NEVE, to avoid long hardware development and deployment cycles, we introduce virtual hardware to mimic the new architectural features on existing hardware. This makes it possible to evaluate architecture extensions for virtualization using existing hardware in ways that were not previously possible. For virtual-passthrough, we introduce virtual hardware in the form of virtual I/O devices directly provided to nested VMs in a new way resulting in substantial performance gains without needing additional actual hardware features. For DVH, we generalize the idea of virtual hardware to provide hardware features to VMs in software without requiring any hardware changes, resulting in even greater performance gains. While this dissertation has focused on using virtual hardware to improve performance, I believe there are many other opportunities for using virtual hardware, including potentially improving the security of computing systems.

Bibliography

[1] Keith Adams and Ole Agesen. “A Comparison of Software and Hardware Techniques for x86 Virtualization.” In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2006)*. San Jose, CA, 2006, pp. 2–13.

[2] Ole Agesen, Jim Mattson, Radu Rugina, and Jeffrey Sheldon. “Software Techniques for Avoiding Hardware Virtualization Exits.” In: *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC 2012)*. Boston, MA, June 2012, pp. 373–385.

[3] Irfan Ahmad, Ajay Gulati, and Ali Mashtizadeh. “vIC: Interrupt Coalescing for Virtual Machine Storage Device IO.” In: *Proceedings of the 2011 USENIX Annual Technical Conference (USENIX ATC 2011)*. Portland, OR, June 2011, pp. 45–58.

[4] Alexander Graf, Joerg Roedel. “Nesting the Virtualized World.” In: *Linux Plumbers Conference*. Sept. 2009.

[5] AMD Corporation. *AMD I/O virtualization technology (IOMMU) specification. Revision 3.00.* Dec. 2016.

[6] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. “vIOMMU: Efficient IOMMU Emulation.” In: *Proceedings of the 2011 USENIX Annual Technical Conference (USENIX ATC 2011)*. Portland, OR, June 2017, pp. 105–121.

[7] Nadav Amit, Muli Ben-Yehuda, and Ben-Ami Yassour. “IOMMU: Strategies for Mitigating the IOTLB Bottleneck.” In: *6th Annual Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA 2010)*. Saint-Malo, France, 2010, pp. 256–274.

[8] Jeremy Andrus, Christoffer Dall, Alex Van’t Hof, Oren Laadan, and Jason Nieh. “Cells: A Virtual Mobile Smartphone Architecture.” In: *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP 2011)*. Cascais, Portugal, Oct. 2011, pp. 173–187.

[9] ARM Ltd. *ARM Generic Interrupt Controller Architecture version 2.0 ARM IHI 0048B*. June 2011.

[10] Arm Ltd. *Arm Architecture Reference Manual Armv8-A DDI 0487F.b*. Mar. 2020.

[11] Arm Ltd. *Arm Architecture Reference Manual Armv8-A DDI 0487F.b, D5.7.2 Enhanced support for nested virtualization*. Mar. 2020.

[12] Arm Ltd. *Arm Generic Interrupt Controller Architecture Specification GIC architecture version 3 and version 4 ARM IHI 0069F*. Feb. 2020.

[13] Arm Ltd. *Arm System Memory Management Unit Architecture Specification version 3.0, 3.1 and 3.2*. July 2019.

[14] Eric Auger. *[PATCH v8 00/14] ARM SMMUv3 Emulation Support*. QEMU Mailing List. Feb. 2018. URL: `http://lists.gnu.org/archive/html/qemu-arm/2018-02/msg00034.html` (Accessed: Oct. 12, 2020).

[15] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. "Xen and the Art of Virtualization." In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*. Bolton Landing, NY, Oct. 2003, pp. 164–177.

[16] Jeff Barr. *Now Available – Compute-Intensive C5 Instances for Amazon EC2*. AWS News Blog. Nov. 2017. URL: `https://aws.amazon.com/blogs/aws/now-available-compute-intensive-c5-instances-for-amazon-ec2/` (Accessed: Oct. 12, 2020).

[17] Ken Barr, Prashanth Bungale, Stephen Deasy, Viktor Gyuris, Perry Hung, Craig Newell, Harvey Tuch, and Bruno Zoppis. "The VMware Mobile Virtualization Platform: Is That a Hypervisor in Your Pocket?" In: *ACM SIGOPS Operating Systems Review* 44.4 (Dec. 2010), pp. 124–135.

[18] Dina Bass and Ian King. *Microsoft Pledges to Use ARM Server Chips, Threatening Intel's Dominance*. Bloomberg. Mar. 2017. URL: `https://www.bloomberg.com/news/articles/2017-03-08/microsoft-pledges-to-use-arm-server-chips-threatening-intel-s-dominance` (Accessed: Oct. 12, 2020).

[19] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. "The Turtles Project: Design and Implementation of Nested Virtualization." In: *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI 2010)*. Vancouver, Canada, Oct. 2010, pp. 423–436.

[20] Paolo Bonzini. *Re: Migration with directly assigned devices is possible?* KVM Mailing List. Apr. 2018. URL: `https://marc.info/?l=kvm&m=152459004513285&w=2` (Accessed: Oct. 12, 2020).

[21] David Brash. *ARMv8-A Architecture - 2016 Additions*. arm Community. Oct. 2016. URL: https://community.arm.com/groups/processors/blog/2016/10/27/armv8-a-architecture-2016-additions (Accessed: Oct. 12, 2020).

[22] Edouard Bugnion, Jason Nieh, and Dan Tsafrir. *Hardware and Software Support for Virtualization*. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, Feb. 2017.

[23] Cesare Cantu. *Network Interface Card Device Pass-Through with Multiple Nested Hypervisors. US Patent 20140310704A1*. Apr. 2013.

[24] Christopher Dall. "The Design, Implementation, and Evaluation of the Linux ARM Hypervisor." PhD thesis. Columbia University, Feb. 2018.

[25] Sean Christopherson. *KVM: nVMX: Disable intercept for FS/GS base MSRs in vmcs02 when possible*. Linux Kernel Source Tree. 2019. URL: https://github.com/torvalds/linux/commit/d69129b4e46a7b61dc956af038d143eb791f22c7 (Accessed: Oct. 12, 2020).

[26] Citrix. *Citrix and AWS partner to enable application elasticity and scale*. 2020. URL: https://www.citrix.com/global-partners/amazon-web-services/ (Accessed: Oct. 12, 2020).

[27] CloudShare. *Easily create nested virtualization in CloudShare*. 2020. URL: https://www.cloudshare.com/technology/nested-virtualization/ (Accessed: Oct. 12, 2020).

[28] Bob Cmelik and David Keppel. "Shade: A Fast Instruction-set Simulator for Execution Profiling." In: *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1994)*. Nashville, TN, May 1994, pp. 128–137.

[29] Wim Coekaerts. *Linux mainline contains all the Xen code bits for Dom0 and DomU support*. Oracle blog. May 2011. URL: https://blogs.oracle.com/wim/linux-mainline-contains-all-the-xen-code-bits-for-dom0-and-domu-support (Accessed: Oct. 12, 2020).

[30] Christoffer Dall, Shih-Wei Li, Jintack Lim, Jason Nieh, and Georgios Koloventzos. "ARM Virtualization: Performance and Architectural Implications." In: *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA 2016)*. Seoul, South Korea, June 2016, pp. 304–316.

[31] Christoffer Dall, Shih-Wei Li, and Jason Nieh. “Optimizing the Design and Implementation of the Linux ARM Hypervisor.” In: *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC 2017)*. Santa Clara, CA, July 2017, pp. 221–234.

[32] Christoffer Dall and Jason Nieh. “KVM for ARM.” In: *Proceedings of the 12th Annual Linux Symposium*. Ottawa, Canada, July 2010, pp. 45–56.

[33] Christoffer Dall and Jason Nieh. *KVM/ARM: Experiences Building the Linux ARM Hypervisor*. Technical Report CUCS-010-13. Department of Computer Science, Columbia University, June 2013.

[34] Christoffer Dall and Jason Nieh. “KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor.” In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2014)*. Salt Lake City, UT, Mar. 2014, pp. 333–347.

[35] Jiun-Hung Ding, Chang-Jung Lin, Ping-Hao Chang, Chieh-Hao Tsang, Wei-Chung Hsu, and Yeh-Ching Chung. “ARMvisor: System Virtualization for ARM.” In: *Proceedings of the 14th Annual Linux Symposium*. Ottawa, Canada, July 2012, pp. 93–107.

[36] Yaozu Dong, Yu Chen, Zhenhao Pan, Jinquan Dai, and Yunhong Jiang. “ReNIC: Architectural Extension to SR-IOV I/O Virtualization for Efficient Replication.” In: *ACM Transactions on Architecture and Code Optimization* 8.4 (Jan. 2012), 40:1–40:22.

[37] Yaozu Dong, Dongxiao Xu, Yang Zhang, and Guangdeng Liao. “Optimizing network I/O virtualization with efficient interrupt coalescing and virtual receive side scaling.” In: *Proceedings of the IEEE International Conference on Cluster Computing*. Austin, TX, Sept. 2011, pp. 26–34.

[38] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. “High Performance Network Virtualization with SR-IOV.” In: *Journal of Parallel and Distributed Computing* 72.11 (Nov. 2012), pp. 1471–1480.

[39] DPDK. *Data Plane Development Kit*. 2019. URL: `https://dpdk.org/` (Accessed: Oct. 12, 2020).

[40] DPDK. *Poll Mode Driver for Emulated Virtio NIC*. 2015. URL: `https://doc.dpdk.org/guides/nics/virtio.html` (Accessed: Oct. 12, 2020).

[41] Dmitry Duplyakin et al. “The Design and Operation of CloudLab.” In: *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC 2019)*. Renton, WA, July 2019, pp. 1–14.

[42] Paul Durrant. *Windows PV Drivers*. Xen Project. 2018. URL: https://xenproject.org/developers/teams/windows-pv-drivers/ (Accessed: Oct. 12, 2020).

[43] Joy Fan. *Nested Virtualization in Azure*. Azure Blog. July 2017. URL: https://azure.microsoft.com/en-us/blog/nested-virtualization-in-azure/ (Accessed: Oct. 12, 2020).

[44] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. "Microkernels Meet Recursive Virtual Machines." In: *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI 1996)*. Seattle, WA, 1996, pp. 137–151.

[45] Google Cloud. *Enabling Nested Virtualization for VM Instances*. Apr. 2018. URL: https://cloud.google.com/compute/docs/instances/enable-nested-virtualization-vm-instances (Accessed: Oct. 12, 2020).

[46] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. "ELI: Bare-metal Performance for I/O Virtualization." In: *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*. London, England, UK, 2012, pp. 411–422.

[47] Abel Gordon, Nadav Har'El, Alex Landau, Muli Ben-Yehuda, and Avishay Traeger. "Towards Exitless and Efficient Paravirtual I/O." In: *Proceedings of the 5th Annual International Systems and Storage Conference (SYSTOR 2012)*. Haifa, Israel, June 2012, 10:1–10:6.

[48] Green Hills Software. *INTEGRITY Secure Virtualization*. Jan. 2014. URL: http://www.ghs.com/products/rtos/integrity_virtualization.html (Accessed: Oct. 12, 2020).

[49] Matthew Gretton-Dann. *Introducing 2017's extensions to the Arm Architectures*. arm Community. Nov. 2017. URL: https://community.arm.com/processors/b/blog/posts/introducing-2017s-extensions-to-the-arm-architecture (Accessed: Oct. 12, 2020).

[50] HaiBing Guan, YaoZu Dong, RuHui ma, Dongxiao Xu, Yang Zhang, and Jian Li. "Performance Enhancement for Network I/O Virtualization with Efficient Interrupt Coalescing and Virtual Receive-Side Scaling." In: *IEEE Transactions on Parallel and Distributed Systems* 24.6 (June 2013), pp. 1118–1128.

[51] Stefan Hajnoczi. "An Updated Overview of the QEMU Storage Stack." In: *LinuxCon Japan 2011*. Yokohama, Japan, June 2011.

[52] Stefan Hajnoczi. *QEMU Internals: vhost architecture*. Sept. 2011. URL: http://blog.vmsplice.net/2011/09/qemu-internals-vhost-architecture.html (Accessed: Oct. 12, 2020).

[53] Nadav Har'El, Abel Gordon, Alex Landau, Muli Ben-Yehuda, Avishay Traeger, and Razya Ladelsky. "Efficient and Scalable Paravirtual I/O System." In: *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC 2013)*. San Jose, CA, June 2013, pp. 231–242.

[54] Joo-Young Hwang, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. "Xen on ARM: System Virtualization using Xen Hypervisor for ARM-based Secure Mobile Phones." In: *Proceedings of the 2008 5th IEEE Consumer Communications and Networking Conference*. Las Vegas, NV, Jan. 2008, pp. 257–261.

[55] Intel Corporation. *4th Generation Intel Core vPro Processors with Intel VMCS Shadowing*. 2013. URL: http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-vmcs-shadowing-paper.pdf (Accessed: Oct. 12, 2020).

[56] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, 325462-044US*. Aug. 2012.

[57] Intel Corporation. *Intel Virtualization Technology for Directed I/O, D51397-011, Rev. 3.1*. June 2019.

[58] Jeffrey Fulmer. *Siege Home*. Joe Dog Software. Jan. 2012. URL: https://www.joedog.org/siege-home/ (Accessed: Oct. 12, 2020).

[59] Richard WM Jones. *Super-nested KVM*. 2014. URL: https://rwmj.wordpress.com/2014/07/03/super-nested-kvm/ (Accessed: Oct. 12, 2020).

[60] Rick Jones. *Netperf*. Dec. 2010. URL: https://github.com/HewlettPackard/netperf (Accessed: Oct. 12, 2020).

[61] Asim Kadav and Michael M. Swift. "Live Migration of Direct-access Devices." In: *Proceedings of the 1st Workshop on I/O Virtualization*. WIOV 2008. San Diego, CA, 2008.

[62] Bernhard Kauer, Paulo Verissimo, and Alysson Bessani. "Recursive Virtual Machines for Advanced Security Mechanisms." In: *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DNSW 2011)*. Hong Kong, China, June 2011, pp. 117–122.

[63] KernelNewbies. *Linux 3.10 Change log - Timerless multitasking*. Dec. 2017. URL: https://kernelnewbies.org/Linux_3.10 (Accessed: Oct. 12, 2020).

[64] Yossi Kuperman, Eyal Moscovici, Joel Nider, Razya Ladelsky, Abel Gordon, and Dan Tsafrir. "Paravirtual Remote I/O." In: *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2016)*. Atlanta, Georgia, USA, Apr. 2016, pp. 49–65.

[65] KVM contributors. *KVM-unit-tests*. Aug. 2020. URL: https://www.linux-kvm.org/index.php?title=KVM-unit-tests&oldid=174016 (Accessed: Oct. 12, 2020).

[66] KVM contributors. *Tuning KVM*. June 2018. URL: https://www.linux-kvm.org/index.php?title=Tuning_KVM&oldid=173911 (Accessed: Oct. 12, 2020).

[67] KVM contributors. *Virtio*. Oct. 2016. URL: https://www.linux-kvm.org/index.php?title=Virtio&oldid=173787 (Accessed: Oct. 12, 2020).

[68] KVM contributors. *WindowsGuestDrivers/Download Drivers*. Nov. 2018. URL: https://www.linux-kvm.org/index.php?title=WindowsGuestDrivers/Download_Drivers&oldid=173940 (Accessed: Oct. 12, 2020).

[69] Joshua LeVasseur, Volkmar Uhlig, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. *Pre-Virtualization: Slashing the Cost of Virtualization*. Technical Report 2005-30. Fakultat fur Informatik, Universitat Karlsruhe (TH), Nov. 2005.

[70] Shih-Wei Li, John S. Koh, and Jason Nieh. "Protecting Cloud Virtual Machines from Commodity Hypervisor and Host Operating System Exploits." In: *Proceedings of the 28th USENIX Security Symposium (USENIX Security 2019)*. Santa Clara, CA, Aug. 2019, pp. 1357–1374.

[71] Wanpeng Li. *KVM: X86: Provide a capability to disable HLT intercepts*. Linux Kernel Source Tree. 2018. URL: https://github.com/torvalds/linux/commit/caa057a2cad647fb368a12c8e6c410ac4c28e063 (Accessed: Oct. 12, 2020).

[72] Cunming Liang and Tiwei Bie. "vdpa: vhost-mdev as a New vhost Protocol Transport." In: *KVM Forum 2018*. Edinburgh, Scotland, UK, 2018.

[73] Jin Tack Lim and Jason Nieh. "Optimizing Nested Virtualization Performance Using Direct Virtual Hardware." In: *Proceedings of the 25th International Conference*

on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2020). Lausanne, Switzerland, Mar. 2020, pp. 557–574.

[74] Jintack Lim. *[RFC 00/55] Nested Virtualization on KVM/ARM*. Linux Kernel Mailing List. Jan. 2017. URL: https://lore.kernel.org/patchwork/cover/748963/ (Accessed: Oct. 12, 2020).

[75] Jintack Lim, Christoffer Dall, Shih-Wei Li, Jason Nieh, and Marc Zyngier. "NEVE: Nested Virtualization Extensions for ARM." In: *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP 2017)*. Shanghai, China, Oct. 2017, pp. 201–217.

[76] Jiuxing Liu. "Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support." In: *Proceedings of the 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS 2010)*. Apr. 2010, pp. 1–12.

[77] Jiuxing Liu, Wei Huang, Bulent Abali, and Dhabaleswar K. Panda. "High Performance VMM-bypass I/O in Virtual Machines." In: *Proceedings of the 2006 USENIX Annual Technical Conference (USENIX ATC 2006)*. Boston, MA, May 2006, pp. 29–42.

[78] Jim Mattson. *[PATCH v5 2/2] kvm: nVMX: Introduce KVM_CAP_NESTED_STATE*. Linux Kernel Mailing List. July 2018. URL: https://patchwork.kernel.org/patch/10516673/ (Accessed: Oct. 12, 2020).

[79] Paul E. McKenney. *NO_HZ: Reducing Scheduling-Clock Ticks*. Linux Kernel Source Tree. May 2017. URL: https://www.kernel.org/doc/Documentation/timers/NO_HZ.txt (Accessed: Oct. 12, 2020).

[80] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel. "Diagnosing Performance Overheads in the Xen Virtual Machine Environment." In: *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE 2005)*. Chicago, IL, June 2005, pp. 13–23.

[81] Aravind Menon, Simon Schubert, and Willy Zwaenepoel. "TwinDrivers: Semi-automatic Derivation of Fast and Safe Hypervisor Network Drivers from Guest OS Drivers." In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2009)*. Washington, DC, Mar. 2009, pp. 301–312.

[82] Microsoft. *Virtualization-based Security (VBS)*. Oct. 2017. URL: https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs (Accessed: Oct. 12, 2020).

[83] Microsoft. *Windows XP Mode*. 2009. URL: https://www.microsoft.com/en-us/download/details.aspx?id=8002 (Accessed: Oct. 12, 2020).

[84] Timothy Prickett Morgan. *Azure Stack Gives Microsoft Leverage Over AWS, Google*. The Next Platform. Jan. 2016. URL: https://www.nextplatform.com/2016/01/26/azure-stack-gives-microsoft-leverage-over-aws-google/ (Accessed: Oct. 12, 2020).

[85] Zhenhao Pan, Yaozu Dong, Yu Chen, Lei Zhang, and Zhijiao Zhang. "CompSC: Live Migration with Pass-through Devices." In: *Proceedings of the 8th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2012)*. London, England, UK, 2012, pp. 109–120.

[86] PCI-SIG. *Address Translation Services Revision 1.1*. Dec. 2009.

[87] Himanshu Raj and Karsten Schwan. "High Performance and Scalable I/O Virtualization via Self-virtualized Devices." In: *Proceedings of the 16th International Symposium on High Performance Distributed Computing (HPDC 07)*. Monterey, CA, June 2007, pp. 179–188.

[88] Kaushik Kumar Ram, Jose Renato Santos, Yoshio Turner, Alan L. Cox, and Scott Rixner. "Achieving 10 Gb/s Using Safe and Transparent Network Interface Virtualization." In: *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2019)*. Washington, DC, 2009, pp. 61–70.

[89] Ravello Community. *Nested virtualization: How to run nested KVM on AWS or Google Cloud*. Ravello Blog. Jan. 2016. URL: https://blogs.oracle.com/ravello/run-nested-kvm-on-aws-google (Accessed: Oct. 12, 2020).

[90] Reuters. *Cloud companies consider Intel rivals after security flaws found*. CNBC News. Jan. 2018. URL: https://www.cnbc.com/2018/01/10/cloud-companies-consider-intel-rivals-after-security-flaws-found.html (Accessed: Oct. 12, 2020).

[91] RISC-V International. *RISC-V*. 2020. URL: http://www.riscv.org (Accessed: Oct. 12, 2020).

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

[93] Rusty Russell, Yanmin Zhang, Ingo Molnar, and David Sommerseth. *Improve hackbench*. Linux Kernel Mailing List. Jan. 2008. URL: http://people.

redhat.com/mingo/cfs-scheduler/tools/hackbench.c (Accessed: Oct. 12, 2020).

[94] Jose Renato Santos, Yoshio Turner, G. Janakiraman, and Ian Pratt. “Bridging the Gap Between Software and Hardware Techniques for I/O Virtualization.” In: *Proceedings of the 2008 USENIX Annual Technical Conference (USENIX ATC 2008)*. Boston, MA, June 2008, pp. 29–42.

[95] Simon Sharwood. *AWS adopts home-brewed KVM as new hypervisor*. The Register. Nov. 2017. URL: https://www.theregister.co.uk/2017/11/07/aws_writes_new_kvm_based_hypervisor_to_make_its_cloud_go_faster/ (Accessed: Oct. 12, 2020).

[96] Paul Sim. *KVM Performance Optimization*. 2013. URL: https://www.slideshare.net/janghoonsim/kvm-performance-optimization-for-ubuntu (Accessed: Oct. 12, 2020).

[97] Standard Performance Evaluation Corporation. *SPECjvm2008*. Aug. 2020. URL: https://www.spec.org/jvm2008 (Accessed: Oct. 12, 2020).

[98] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. “Virtualizing I/O Devices on VMware Workstation’s Hosted Virtual Machine Monitor.” In: *Proceedings of the 2001 USENIX Annual Technical Conference (USENIX ATC 2001)*. Boston, MA, June 2001, pp. 1–14.

[99] SUSE. *Disk Cache Modes*. SUSE Product Documentation. Sept. 2020. URL: https://documentation.suse.com/sles/12-SP4/html/SLES-all/cha-cachemodes.html (Accessed: Oct. 12, 2020).

[100] The Apache Software Foundation. *ab - Apache HTTP server benchmarking tool*. Apr. 2015. URL: http://httpd.apache.org/docs/2.4/programs/ab.html (Accessed: Oct. 12, 2020).

[101] Michael S. Tsirkin. *vhost_net: a kernel-level virtio server*. Linux Kernel Mailing List. Aug. 2009. URL: https://lwn.net/Articles/346267/ (Accessed: Oct. 12, 2020).

[102] Michael S. Tsirkin, Cornelia Huck, and Pawel Moll. *Virtual I/O Device (VIRTIO) Version 1.0 virtio-v1.0-cs04*. Mar. 2016.

[103] Cheng-Chun Tu, Michael Ferdman, Chao-tung Lee, and Tzi-cker Chiueh. “A Comprehensive Implementation and Evaluation of Direct Interrupt Delivery.” In: *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2015)*. Istanbul, Turkey, Mar. 2015, pp. 1–15.

[104] Prashant Varanasi and Gernot Heiser. "Hardware-supported Virtualization on ARM." In: *Proceedings of the 2nd Asia-Pacific Workshop on Systems (APSys 2011)*. Shanghai, China, July 2011, 11:1–11:5.

[105] Lluis Vilanova, Nadav Amit, and Yoav Etsion. "Using SMT to Accelerate Nested Virtualization." In: *Proceedings of the 46th International Symposium on Computer Architecture (ISCA 2019)*. Phoenix, Arizona, June 2019, pp. 750–761.

[106] VMware. *VMware Tools Device Drivers*. VMware Docs. May 2019. URL: `https://docs.vmware.com/en/VMware-Tools/10.1.0/com.vmware.vsphere.vmwaretools.doc/GUID-6994A5F9-B62B-4BF1-99D8-E325874A4C7A.html` (Accessed: Oct. 12, 2020).

[107] Carl Waldspurger and Mendel Rosenblum. "I/O Virtualization." In: *Communications of the ACM* 55.1 (Jan. 2012), pp. 66–73.

[108] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. ""Scale and Performance in the Denali Isolation Kernel"." In: *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*. Boston, MA, 2002, pp. 195–209.

[109] Dan Williams, Yaohui Hu, Umesh Deshpande, Piush K. Sinha, Nilton Bila, Kartik Gopalan, and Hani Jamjoom. "Enabling Efficient Hypervisor-as-a-Service Clouds with Ephemeral Virtualization." In: *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2016)*. Atlanta, GA, Apr. 2016, pp. 79–92.

[110] Dan Williams, Hani Jamjoom, and Hakim Weatherspoon. "The Xen-Blanket: Virtualize Once, Run Everywhere." In: *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys 2012)*. Bern, Switzerland, Apr. 2012, pp. 113–126.

[111] Alex Williamson. "VFIO: A user's perspective." In: *KVM Forum 2012*. Feb. 2012.

[112] Alex Williamson. *vfio/pci: Improve extended capability comments, skip masked caps*. QEMU Source Tree. Feb. 2017. URL: `https://github.com/qemu/qemu/commit/d0d1cd70d10639273e2a23870e7e7d80b2bc4e21` (Accessed: Oct. 12, 2020).

[113] Alex Williamson, Alexey Kardashevskiy, Linus Torvalds, Zi Shen Lim, Gavin Shan, and Mauro Carvalho Chehab. *VFIO - "Virtual Function I/O"*. Linux Kernel Documentation. May 2017. URL: `https://www.kernel.org/doc/Documentation/vfio.txt` (Accessed: Oct. 12, 2020).

[114] Paul Willmann, Scott Rixner, and Alan L. Cox. "Protection Strategies for Direct Access to Virtualized I/O Devices." In: *Proceedings of the 2008 USENIX Annual Technical Conference (USENIX ATC 2008)*. Boston, MA, June 2008, pp. 15–28.

[115] Paul Willmann, Jeffrey Shafer, David Carr, Aravind Menon, Scott Rixner, Alan L. Cox, and Willy Zwaenepoel. "Concurrent Direct Network Access for Virtual Machine Monitors." In: *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA 2007)*. Scottsdale, AZ, Feb. 2007, pp. 306–317.

[116] Xen Project Wiki. *Nested Virtualization in Xen*. July 2018. URL: `https://wiki.xenproject.org/wiki/Nested_Virtualization_in_Xen` (Accessed: Oct. 12, 2020).

[117] Xen Project Wiki. *Network Throughput and Performance Guide*. Apr. 2014. URL: `http://wiki.xen.org/wiki/Network_Throughput_and_Performance_Guide` (Accessed: Oct. 12, 2020).

[118] Xen Project Wiki. *Xen ARM with Virtualization Extensions*. June 2020. URL: `http://wiki.xenproject.org/wiki/Xen_ARM_with_Virtualization_Extensions` (Accessed: Oct. 12, 2020).

[119] Xen Project Wiki. *Xen PCI Passthrough*. Nov. 2019. URL: `https://wiki.xen.org/wiki/Xen_PCI_Passthrough` (Accessed: Oct. 12, 2020).

[120] Cong Xu, Sahan Gamage, Hui Lu, Ramana Kompella, and Dongyan Xu. "vTurbo: Accelerating Virtual Machine I/O Processing Using Designated Turbo-sliced Core." In: *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC 2013)*. San Jose, CA, 2013, pp. 243–254.

[121] Peter Xu. *[PATCH v3 0/2] iommu/vt-d: Fix mapping PSI missing for iommu_map()*. Linux Kernel Mailing List. May 2018. URL: `https://lists.linuxfoundation.org/pipermail/iommu/2018-May/027479.html`.

[122] Peter Xu. *[PATCH v4 0/9] intel-iommu: nested vIOMMU, cleanups, bug fixes*. QEMU Mailing List. May 2018. URL: `http://lists.nongnu.org/archive/html/qemu-devel/2018-05/msg04291.html` (Accessed: Oct. 12, 2020).

[123] Peter Xu. *intel-iommu: rework the page walk logic*. QEMU Source Tree. May 2018. URL: `https://github.com/qemu/qemu/commit/63b88968f139b6a77f2f81e6f1eedf70c0170a85` (Accessed: Oct. 12, 2020).

[124] Xin Xu and Bhavesh Davda. "A Hypervisor Approach to Enable Live Migration with Passthrough SR-IOV Network Devices." In: *ACM SIGOPS Operating Systems Review* 51.1 (Sept. 2017), pp. 15–23.

[125] Edwin Zhai, Gregory D. Cummings, and Yaozu Dong. "Live Migration with Passthrough Device for Linux VM." In: *Proceedings of the 10th Linux Symposium*. Ottawa, Canada, July 2008, pp. 261–267.

[126] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. "CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization." In: *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP 2011)*. Cascais, Portugal, Oct. 2011, pp. 203–216.

[127] Marc Zyngier. *[PATCH v2 00/94] KVM: arm64: ARMv8.3/8.4 Nested Virtualization support*. Linux Kernel Mailing List. Feb. 2020. URL: `https://lore.kernel.org/linux-arm-kernel/20200211174938.27809-1-maz@kernel.org/` (Accessed: Oct. 12, 2020).