Optimizing Nested Virtualization Performance Using Direct Virtual Hardware

Abstract

Nested virtualization, running virtual machines and hypervisors on top of other virtual machines and hypervisors, is increasingly important because of the need to deploy virtual machines running software stacks on top of virtualized cloud infrastructure. However, performance remains a key impediment to further adoption as application workloads can perform many times worse than native execution. To address this problem, we introduce DVH (Direct Virtual Hardware), a new approach that enables a host hypervisor, the hypervisor that runs directly on the hardware, to directly provide virtual hardware to nested virtual machines without the intervention of multiple levels of hypervisors. We introduce four DVH mechanisms, virtual-passthrough, 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 the Linux KVM hypervisor. Our experimental results show that DVH can provide near native execution speeds and improve KVM performance by more than an order of magnitude on real application workloads.

CCS Concepts • **Software and its engineering → Virtual machines**; *Operating systems*; • **Computer systems organization** → *Cloud computing*; *Architectures.*

Keywords nested virtualization; hypervisors; I/O virtualization; performance

1 Introduction

Nested virtualization involves running multiple levels of hypervisors to support running virtual machines (VMs) inside VMs. It 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 [12, 21, 23, 40, 43]. Furthermore, operating systems (OSes) including Linux and Windows have built-in hypervisors to support legacy applications [37] and enhance security [38]; these OS features require nested virtualization support to run in VMs. However, poor nested virtualization performance remains a key issue for many application workloads and an impediment to further adoption.

Some approaches exist for addressing parts of this problem, such as device passthrough for improving I/O performance. Device 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 [6, 9]. For example, the physical device can deliver data directly to the nested VM. However, device passthrough 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 [7, 57], I/O device consolidation, and various VM memory optimizations [8]. Many of these features, especially migration [41], are essential for cloud computing deployments. Furthermore, device passthrough requires additional hardware support such as physical Input/Output Memory Management Units (IOMMUs) and Single-Root I/O Virtualization (SR-IOV). Because of these disadvantages, paravirtual I/O devices are more commonly used in VM deployments. Unfortunately, nested virtualization with virtual I/O devices, including paravirtual I/O devices, incurs high overhead.

We introduce Direct Virtual Hardware (DVH), a new approach to enhancing nested virtualization performance in which the host hypervisor, the hypervisor that runs natively on the hardware, directly provides virtual hardware to nested VMs. Nested VMs can then interact with the virtual hardware without the intervention of multiple layers of hypervisors. A

key characteristic of DVH that makes this possible is that the virtual hardware is provided by a different hypervisor layer other than the one responsible for managing the nested VM. This also has implications for the virtual hardware design. 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. DVH makes it possible to support novel virtualization optimizations only in software, and even introduces new virtual hardware capabilities that are not natively supported by the 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. We present four DVH mechanisms for the x86 architecture: virtual-passthrough, virtual timers, virtual inter-processor interrupts (IPIs), and virtual idle.

Virtual-passthrough is a novel yet simple technique for boosting I/O performance for nested virtualization. Virtual-passthrough is similar to device passthrough, but assigns virtual I/O devices to nested VMs instead of physical devices. 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. 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 [1] are made available and used by intervening hypervisors to provide necessary mappings between different guest physical address spaces to support transferring data between nested VM memory and the virtual I/O device provided by the host hypervisor. 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 does not require hardware support such as physical IOMMUs or SR-IOV, and easily supports important virtualization features such as migration.

Virtual timers reduce the latency of programming CPU timers. On architectures like x86 that do not provide a separate timer for VMs, programming a timer from a VM causes a trap to the hypervisor which needs to emulate the hardware behavior. This results in multiple levels of hypervisor interventions for nested virtualization. Virtual timers appear to intervening hypervisors as an additional hardware timer capability just for VMs to use, but require no additional hardware support beyond existing CPU hardware timers. They do not need to be emulated by multiple layers of hypervisors. The host hypervisor provides virtual timer emulation including transparently remapping hardware timers used by nested VMs to the virtual timers and accounting for timer offset differences.

Virtual IPIs reduce the latency of sending IPIs between virtual CPUs used by nested VMs. An IPI is a special interrupt that allows one CPU to interrupt another by setting a register to indicate the type of message and the destination CPU. Hypervisors prevent VMs from directly configuring the register to raise IPIs to preserve VM isolation and hypervisor control. Programming an IPI from a VM causes a trap to the hypervisor which needs to emulate the hardware behavior, resulting in multiple levels of IPI emulation for nested virtualization. Virtual IPIs make use of a virtual IPI register which appears to intervening hypervisors as an additional hardware capability just for VMs to use. Nested VMs no longer need to trap to guest hypervisors to send IPIs. Guest hypervisors pass along virtual CPU mappings to the virtual hardware in the host hypervisor, enabling the host hypervisor to translate the nested VM IPI destination CPU to the correct physical CPU.

Virtual idle reduces the latency of switching to and from low-power mode. OSes switch to low-power mode when there are no jobs to run. Entering and exiting low-power mode in a VM is emulated by the hypervisor, resulting in multiple levels of hypervisor interventions for nested virtualization. Virtual idle leverages existing architectural support to configure guest hypervisors not to trap the instruction for entering low-power mode so that only the host hypervisor provides low-power mode emulation similar to non-nested virtualization.

We have implemented DVH in the Linux KVM hypervisor and evaluated its performance. Our results show that DVH can improve KVM performance by more than an order of magnitude when running real application workloads using nested virtualization. In many cases, DVH makes nested virtualization overhead similar to that of non-nested virtualization 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.

2 Background

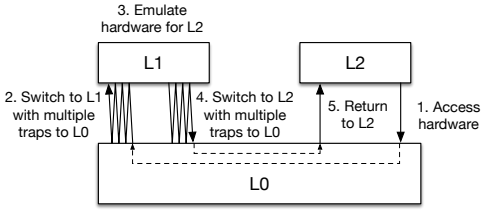
Nested virtualization is the ability to run multiple levels of VMs. Non-nested virtualization runs a hypervisor on physical hardware and provides a virtual execution environment similar to the underlying hardware for the VM. This allows a standard OS designed to run on physical hardware to run without modifications inside the VM. With nested virtualization, the hypervisor must support running another hypervisor within the VM, which can in turn run another VM.

We refer to the host hypervisor as the first hypervisor that runs directly on the hardware, and the guest hypervisor as the hypervisor running inside a VM. 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,

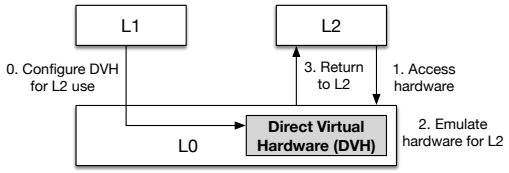
the guest OS or hypervisor as the L1 guest or hypervisor, the guest OS or hypervisor running on top of the L1 hypervisor as the L2 guest or hypervisor, and so on.

Exits from a VM to the hypervisor are the main reason for the overhead in virtualization due to the time spent in the hypervisor [24] and cache pollution [32]. VMs often exit to the hypervisor because the guest OS in the VM is accessing some hardware resource used by both hypervisors and guest OSes, so the VM cannot be allowed to directly configure and manipulate the hardware. An exit occurs to the hypervisor because the hypervisor needs to emulate the hardware behavior for the VM to ensure the hypervisor retains control of the hardware and provides VM isolation. With multiple levels of virtualization, each hypervisor is responsible for emulating the hardware behavior for the VMs that it runs. For example, an L2 guest would exit to the L1 hypervisor, which would be responsible for emulating the hardware behavior for the L2 VM. The L2 guest and L1 hypervisor are encapsulated in an L1 VM, which is then managed by the L0 hypervisor, but the L0 hypervisor does not have full visibility into the internal operation of the L1 VM, including the details of how the L1 hypervisor manages its L2 guest.

Modern hardware such as x86 and ARM systems provides a single-level of architectural support for virtualization, only allowing the host hypervisor to directly use virtualization hardware support [3, 26]. Exits from a nested VM at any virtualization level do not go directly to its respective hypervisor but instead must go first to the host hypervisor, which then can forward the exit to the guest hypervisor to handle. For example, an exit from an L3 guest does not go directly to the L2 hypervisor but instead will first go to the L0 hypervisor, which will forward it to the L1 hypervisor, which will forward it to the L2 hypervisor via the L0 hypervisor to be handled. Guest hypervisors cannot directly use virtualization hardware support, which results in exits from the guest hypervisor to the host hypervisor so it can emulate the virtualization hardware. Because of the need for emulation, virtualization operations performed by the guest hypervisors are much more expensive than those performed by the host hypervisor, which can directly use the hardware. Each exit from a nested VM, therefore, can result in many more exits due to virtualization level switches and further virtualization operation emulation, causing a dramatic increase in virtualization overhead due to exit multiplication [6, 36], depicted in Figure 1a.



(a) L2 hardware access without DVH causing exit multiplication



(b) L2 hardware access with DVH

Figure 1. Hardware access from nested VM

3 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 to exit to guest hypervisors for emulating the respective hardware behavior as shown in Figure 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 [27] and NEVE on ARM [36], architecture extensions defer unnecessary traps from the guest hypervisor, resulting in less traps to the host hypervisor in steps 2 and 4 in Figure 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 and 4 in Figure 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 existance 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 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 3.1 to 3.4. We have also directly used DVH mechanisms such as virtual-passthrough on other architectures such as ARM, but omit further details due to space constraints. DVH can be easily used with additional levels of nested virtualization and supports key virtualization features such as live migration, as discussed in Sections 3.5 and 3.6.

3.1 Virtual-passthrough

In the widely-used traditional virtual I/O model, VMs interact with virtual I/O devices provided by the hypervisor; physical I/O devices are not visible to a VM. Each I/O request, such as sending a network packet or reading a file, is trapped to the hypervisor. The hypervisor processes the request in software, typically leveraging underlying physical devices, and sends an interrupt to the VM to notify it when the I/O request has been completed. For nested I/O virtualization, as shown in Figure 2a, the hypervisor at each level provide its own virtual I/O devices to its VMs in software, which is transparent to the underlying hypervisors. However, this cascade of virtual devices requires the hypervisor at each level to emulate the device behavior, resulting in a multitude of exits due to exit multiplication and poor performance. Device passthrough, shown in Figure 2b, directly assigns physical devices to the nested VM to avoid this cost [6, 9], but at the loss of I/O interposition and its benefits [8].

We introduce virtual-passthrough, a DVH technique for boosting I/O performance for nested virtualization. Virtual-passthrough is similar to device 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 2c, 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 hypervisors to provide their own virtual I/O devices, removing expensive guest hypervisor interventions [6, 36] 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

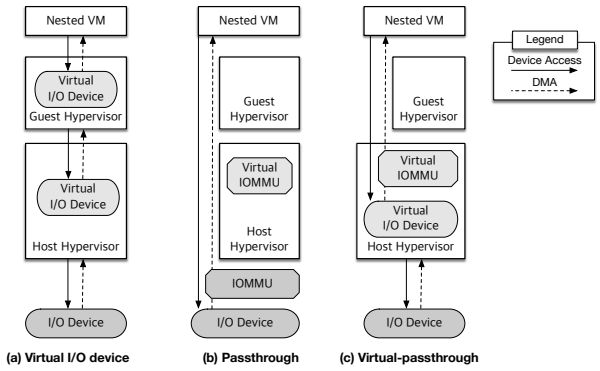


Figure 2. I/O virtualization models

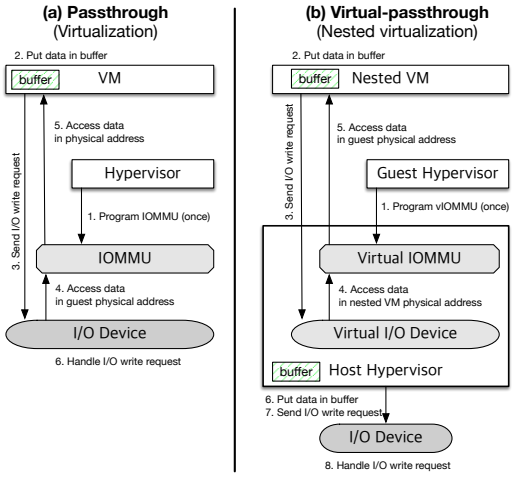


Figure 3. I/O write operation with (virtual) passthrough

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 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 useful for cloud computing deployments where various hypervisors are used on servers [5, 11, 39, 47] and users may freely choose what guest hypervisors and OSes they want to use.

System configuration Virtual-passthrough requires configuration changes in how devices are managed and used, but requires no implementation changes 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 and passthrough models. Virtual-passthrough configures these components in a different way at each virtualization level from the two models, but does not modify or introduce any additional components. We discuss in turn 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 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 through which the hypervisor passes mapping information from VM physical addresses to host physical addresses to hardware. The hypervisor programs the IOMMU to control what physical memory can be accessed by the physical device to guarantee VM memory safety and isolation. 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 [1]. The guest hypervisor programs the virtual IOMMU to control how memory accesses from the virtual I/O device are mapped to memory regions in the nested VM. Through the memory map information, the host hypervisor can emulate memory accesses from the virtual I/O device to the nested VM to guarantee memory safety and isolation. 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 provides its own virtual I/O device to the nested VM, but simply passes

through device access to the virtual I/O device provided by the 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. As a software-only solution, virtual I/O devices, especially paravirtual I/O devices, may in general use any device interface, but 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. Fortunately, PCI-based virtual I/O devices [45] 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. The device is no different from any other PCI devices from the nested VM's perspective. Like the passthrough model, the nested VM just has to have the correct device driver for the given I/O device. As a result, virtual-passthrough is designed to work transparently with nested VMs without any modifications other than potentially device driver installation.

3.2 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. Using 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 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 [26]. The only additional information needed is the interrupt vector number the nested VM programmed for timer interrupts. Further details are omitted due to space constraints.

3.3 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 [26] 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 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 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

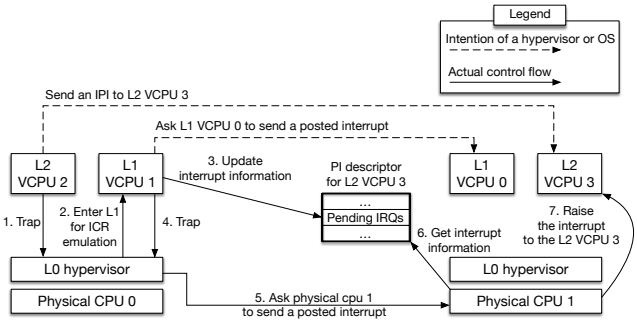


Figure 4. Nested VM IPI delivery

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,

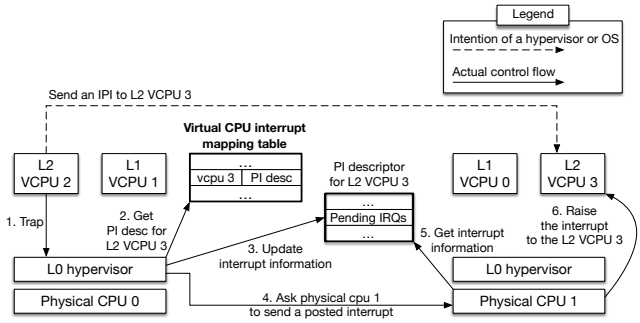


Figure 5. Nested VM IPI delivery with virtual IPIs

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 5 shows the same nested VM IPI delivery example from Figure 4, 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.

3.4 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 excuted 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 3.2. 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 [34] in all hypervisors or using a guest kernel option to poll [51] 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. Using 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.

3.5 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 necessary information to emulate nested VMs. The currently running guest OS in

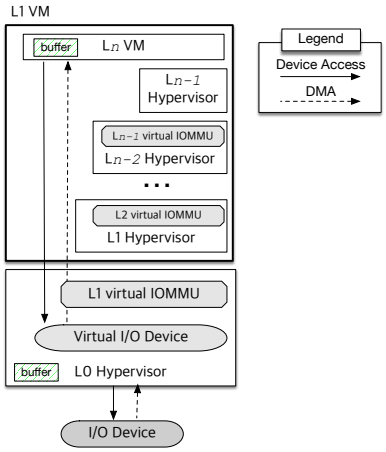


Figure 6. Recursive virtual-passthrough

a nested VM can always use the virtual hardware without trapping to guest hypervisors.

For example, recursive virtual-passthrough can be achieved with a system configuration change. The configuration of the L0 host hypervisor and the nested VM remain the same as with two levels of virtualization as discussed in Section 3.1. The only difference when using more levels of nested virtualization is how the multiple levels of guest hypervisors are configured. 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 virtual IOMMU provided by the host hypervisor (L1 virtual IOMMU) is used when the virtual I/O device accesses L_n memory as shown Figure 6. 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.

As another example, recursive virtual timers also can be achieved in a similar way to recursive virtual-passthrough. 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 will 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-1} hypervisor. For both cases, the respective hypervisor emulates timer functionality for the L_n VM.

3.6 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, migrating a nested VM alone requires some additional support. 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 recopy, the VM can 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 address this problem by leveraging DVH to extend the virtual I/O device provided by the host hypervisor to capture virtual I/O device state and track memory pages dirtied by the virtual I/O device. 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 and device independent, we leverage the extensibility of the PCI standard which provides a mechanism known as capabilities. 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 enable 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 which 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. We assume the same type of host hypervisor is used at the source and destination so that the encapsulated state can be interpreted correctly at the destination. To track memory pages dirtied by the I/O device, we use 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. Leveraging existing functionality minimizes host hypervisor changes as it only requires connecting the migration capability interface to existing functionality. 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.

4 Evaluation

We implemented the four DVH mechanisms in KVM and evaluated their performance. Experiments used x86 server hardware in CloudLab [44], 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 [27] 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 [10]; 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

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 1. Virtualization microbenchmarks

Name	Description
Netperf	`netperf` v2.6.0 [28] 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` [50] 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 default parameters for 30 seconds.
MySQL	`MySQL` v14.14 (distrib 5.5.41) running `SysBench` v.0.4.12 using the default configuration with 200 parallel transactions.
Hackbench	`hackbench` [46] using Unix domain sockets and 100 process groups running with 500 loops.

Table 2. Application benchmarks

network device with vhost-net and the `cache=none` setting for virtual block storage devices [25, 31, 49]. We also provide measurements using Xen 4.10.1 as an x86 guest hypervisor.

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 a 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 48 GB RAM. Two cores and 12 GB RAM were added for the hypervisor at each virtualization level similar to previous work [36, 52, 54] on nested virtualization using multicore processors. We pinned each virtual CPU to a specific physical CPU following best measurement practices [14, 36, 48, 55]. 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 1 and Table 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 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 [29]. Measurements were run using paravirtual I/O, though only DevNotify uses the I/O device. The measurements show more than an order of magnitude increase in cost when run 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 noticably 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 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 nested virtualization level.

	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 3. Microbenchmark performance in CPU cycles

Figure 7 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 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. The native execution results were 45,578 trans/s for Netperf RR, 9,413 Mb/s for Netperf STREAM, 9,414 Mb/s for Netperf MAERTS, 15,469 trans/s for Apache, 354,132 trans/s for Memcached, 4.45 s for MySQL, and 10.36 s for Hackbench.

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. 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 intervention on physical CPUs that run the nested VM.

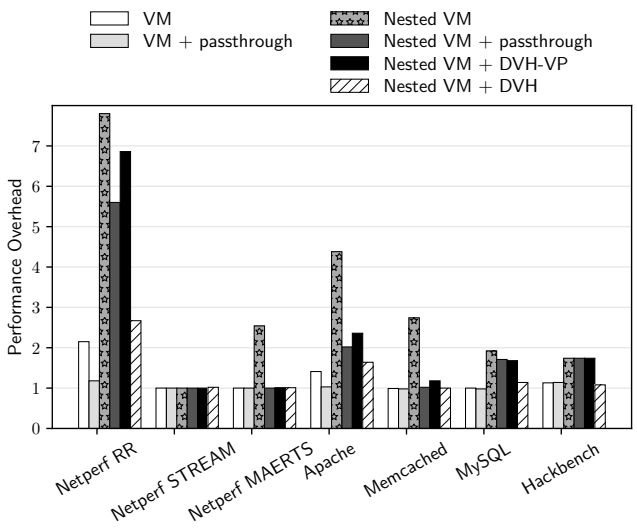


Figure 7. Application performance

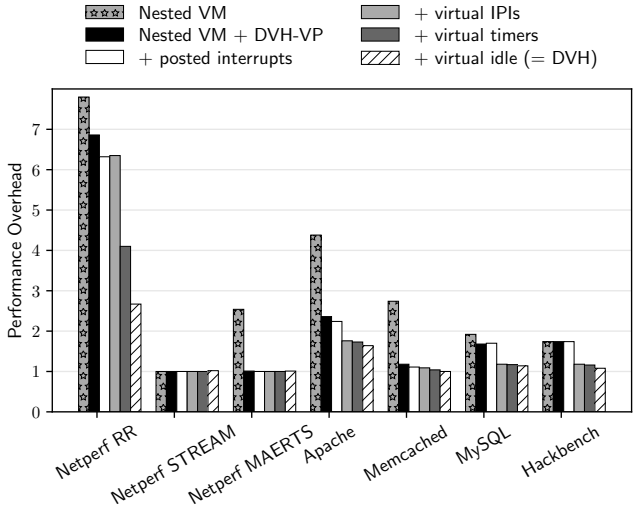


Figure 8. Application performance breakdown

Figure 8 provides a finer granularity breakdown of the nested virtualization performance in Figure 7 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

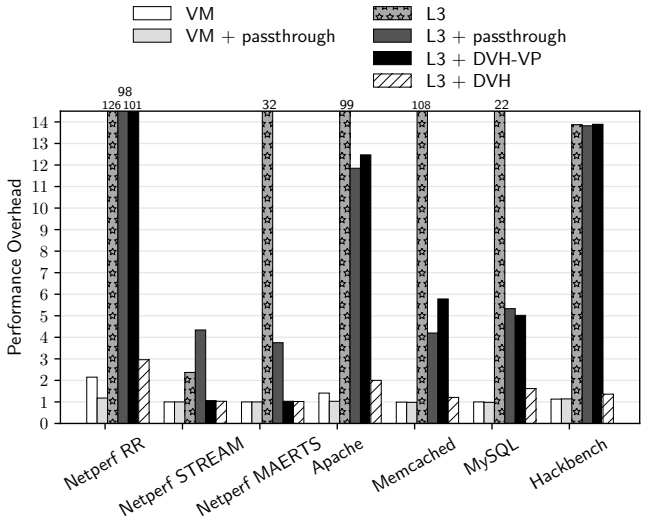


Figure 9. Application performance in L3 VM

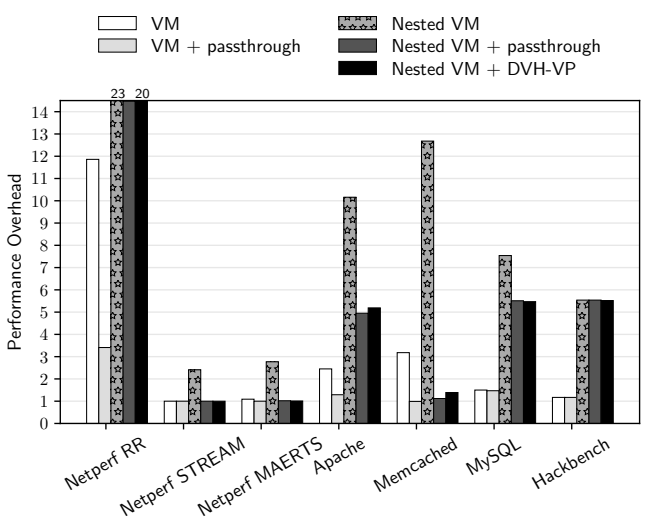


Figure 10. Application performance, Xen on KVM

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 9 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.

Figure 10 shows the same performance measurements as Figure 7, but using Xen instead of KVM. Because nested virtualization support does not work properly in recent Xen versions including the version we used [56], we ran Xen only as the guest hypervisor for the nested VM cases while using KVM as the host hypervisor. Since most DVH techniques typically also require the guest hypervisor to be aware of these virtual hardware mechanisms to use them, we only performed DVH-VP measurements with Xen as virtual-passthrough can be used without any guest hypervisor modifications. Just like the KVM guest hypervisor case, the performance differences among the different I/O configurations is substantial for the nested VM case. Paravirtual I/O performs significantly worse than passthrough for the nested VM case for all application workloads. DVH-VP is able to provide performance similar to passthrough for all workloads and provides substantial gains over the virtual I/O device model, up to an order of magnitude for Memcached. DVH-VP also significantly improved performance on ARM since I/O models are platform-agnostic, but we omit these results due to space constraints.

We also measured the time to migrate VMs and nested VMs running the application workloads by doing live migration between two identical x86 servers on the same subnet. Migration does not work using passthrough, so we compared using paravirtual I/O versus DVH. The default transfer bandwidth configuration of 268 Mbps was used for QEMU for migration to avoid interference with the running workload. Migration times for nested VMs using DVH versus paravirtual I/O were roughly the same, and the times were also roughly the same as migrating VMs. Migrating nested VMs along with their guest hypervisors using DVH was roughly twice as expensive as migrating only the nested VM due to the extra memory state that must transferred. Further details are omitted due to space constraints.

5 Related Work

Modern architectures such as x86 and ARM have been adding more powerful virtualization extensions to enhance VM and nested VM performance [2, 4, 13, 15–17, 26, 27, 33, 36]. Hardware extensions such as APICv on x86 [26] and VGIC on ARM [2, 4] 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 [27] and NEVE on ARM [36] 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.

Denali [53] 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 [22] 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.

Turtles [6] 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.

Dichotomy [54] 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 [51] 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.

Various efforts have tried to compensate for the lack of I/O interposition with passthrough to support live migration. Software-only approaches [30, 42, 58, 59] either do not support unmodified guest OSes or support only specific guest OSes or may lose data due to incomplete tracking of I/O operations. Hardware approaches such as ReNIC [18] propose extending SR-IOV device functionality with a custom interface for device state migration and extending IOMMU functionality for dirty page logging.

Various approaches leverage virtual I/O devices instead of physical I/O devices to balance performance and I/O interposition, such as vDPA [35] and DPDK using virtual I/O devices [19, 20]. Unlike DVH, these approaches do not provide good performance for nested virtualization since the virtual I/O devices that nested VMs use still require expensive guest hypervisor interventions.

6 Conclusions

We introduced DVH, a new approach for directly providing virtual hardware to nested virtual machines without the intervention of multiple levels of hypervisors. We introduce four DVH mechanisms, virtual-passthrough to directly assign virtual I/O devices to nested virtual machines, virtual timers to transparently remap timers used by nested virtual machines to virtual timers provided by the host hypervisor, virtual inter-processor interrupts that can be sent and received directly from one nested virtual machine 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. We have implemented DVH in the Linux KVM hypervisor and show that it can provide more than an order of magnitude better performance than current KVM nested virtualization on real application workloads, in many cases making nested virtualization overhead similar to that of non-nested 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.

7 Acknowledgments

Shih-Wei Li provided helpful feedback on earlier drafts of this paper. Peter Xu and Eric Auger helped with virtual IOMMU internals and configuration in QEMU. This work was supported in part by NSF grants CCF-1918400, CNS-1717801, and CNS-1563555.

A Artifact Appendix

A.1 Abstract

The artifact contains the source code for the DVH implementation in KVM/QEMU on the x86 architecture, a VM disk image, and instructions to measure application benchmark performance. Users can reproduce the results in Figure 7 and Figure 9.

A.2 Artifact check-list (meta-information)

- **Program:** Linux kernel 4.18.0. QEMU 3.1.0.
- **Compilation:** GCC 4.8.4.
- **Run-time environment:** Ubuntu 14.04.
- **Hardware:** Two Intel x86 machines connected by a network.
- **Experiments:** Application benchmarks: Netperf, Apache, Memcached, MySQL, and Hackbench.
- **How much disk space required (approximately)?:** 50 GB to store a VM disk image file.
- **How much time is needed to prepare workflow (approximately)?:** 2 hours.
- **How much time is needed to complete experiments (approximately)?:** 10+ hours.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** GNU GPL v2.
- **Workflow framework used?:** No, but scripts are provided to automate the measurements.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.3555508

A.3 Description

A.3.1 How delivered

The artifact, which is zipped into two files, is available on Zendoo: https://doi.org/10.5281/zenodo.3555508. The source code, scripts, and instructions are zipped into `dvh-asplos-ae.tar.gz`. A VM image is zipped into `ae-guest0.img.bz2`. While we will focus on how to download and use the artifact from Zendoo, the source code, scripts, and instructions are also available on GitHub: https://github.com/columbia/dvh-asplos-ae.

A.3.2 Hardware dependencies

The DVH implementation works on Intel x86 CPUs with support for CPU posted interrupts. Passthrough experiments additionally require CPUs to have Intel VT-d support and the network card to have SR-IOV support.

A.4 Installation

Users need to install software on two Intel x86 machines, a server used for running VMs and a client which imposes workloads on the server machine. Both the server and client should first have Ubuntu 14.04 installed.

Users need to download the source code and scripts zipped into the `dvh-asplos-ae.tar.gz` from Zendoo to the server and the client. While the source code and scripts are also used in the VM, they have already been installed in the provided VM image file under the root directory for each virtualization level. The following is the directory structure of the source code, scripts, and instructions:

- **README.md** This file has detailed instructions to conduct experiments.
- **scripts** This directory has scripts to run experiments.
- **linux** This directory is a git repository having multiple branches of Linux kernel used in this paper.
- **qemu** This directory is a git repository having multiple branches of QEMU used in this paper.

After downloading the source code and scripts, users need to install necessary packages and scripts on the server and the client, which will help users to run VMs and scripts introduced in the later sections. The necessary packages and scripts can be installed using the following commands:

```
# cd scripts
# ./install_scripts.sh
# ./install_packages.sh
```

To conduct experiments, users first need to set up a custom Linux kernel on the server and client. Compiling and installing the Linux kernel is done using the *GNU make* utility on a Linux machine in a standard way. The client kernel needs to be installed and configured only once with the baseline kernel version, `v4.18-base` branch. The server needs to have different kernel versions installed based on the experiment configuration. In addition, QEMU also needs to be compiled using *GNU make* and installed on the server to run VMs.

Users need to have a VM image to run a VM. While users can create their own VM images to run experiments, we provide a pre-configured VM image having another VM image inside which also has another VM image inside to support up to three levels of virtualization. Ubuntu 14.04 is already installed in each VM at each virtualization level, which we can refer to as the L1, L2, and L3 VMs. To obtain the VM image, download the file `ae-guest0.img.bz2` from Zendoo to the server and unzip it using the following command, which requires 50 GB of disk space on the server:

```
# pbzip2 -dk ae-guest0.img.bz2
```

Once the server is set up with the Linux kernel and QEMU, users can run the `run-vm.py` script on the server machine to run VMs so they can be set up to run experiments. The `run-vm.py` script provides three options. The first one is to set up the path to a VM image. The second option is to set up the VM configuration. Available configurations are `base`, which is virtual I/O, `passthrough`, `dvh-vp`, and `dvh`. For setting up the VMs, this configuration option should be set to base, which is the default. The last option is to set a virtualization level from 1 to 3, corresponding to L1 VM, L2 VM, and L3 VM, respectively. The script will automatically run the VMs at the specified level on entering `0`. The following example shows how to run the script with a VM image path set to `/vm/v4.18.img`, a VM configuration of base, and a virtualization level of 2, which will run the L2 VM as well as its required L1 VM:

```
# cd scripts
# ./run-vm.py
--------- VM configurations -------
1. [/vm/v4.18.img] VM Image path
2. [base] VM Configuration
3. [2] Virtualization Level
Enter number to update configuration. Enter 0 to start a VM:
```

To install a Linux kernel version and QEMU on the server and at each virtualization level, start installing them at the lowest level,

the server, and at the subsequent higher levels in order. For example, to setup an L2 DVH configuration, compile and install the Linux kernel on the server with the `v4.18-dvh-L0-asplos` branch, configure kernel parameters to use this kernel, and compile QEMU with the `v3.1.0-dvh` branch on the server. Then reboot the server to use the configured kernel. Next, run `run-vm.py` to run the L1 VM. Since the provided VM image already has the necessary kernels installed, it is only necessary to configure kernel parameters to use the kernel with the `v4.18-dvh-basic-asplos` branch, compile QEMU with the `v3.1.0-base` branch, and terminate the L1 VM. Finally, run `run-vm.py` to run the L2 VM. If the correct kernel version is not being used in the L2 VM, configure kernel parameters, terminate the L2 and L1 VM, and run `run-vm.py` again to boot the L2 VM with the correct kernel version. QEMU is not necessary in the last level VM. Kernel and QEMU versions/configurations for each experiment at each virtualzation level as well as detailed instructions to compile and install the Linux kernel and QEMU can be found in the `README.md`.

A.5 Experiment workflow

We compare application performance on a bare-metal server machine versus VMs running on the server machine at different virtualization levels from 1 to 3. We measure performance of VMs at each level using different configurations - virtual I/O, passthrough I/O, DVH-VP, and DVH. Users run the `run-vm.py` script on the server machine to run VMs, using the second option of the script to select the I/O configuration, as discussed above. For example, to use DVH, the second option would be set to `dvh`. Users do not need to run the script for the native execution measurements.

Once a bare-metal machine or VM is ready on the server, users run the `run-benchmarks.sh` script on the client to start experiments. The script takes a command-line option to indicate the virtualization level to use at the server, L0, which is bare-metal, L1, L2, or L3. When running the script, the user may choose to run all application benchmarks or a subset of them selectively. Once a benchmark is selected, the script will list the benchmark with a `*` on the left. Enter `0` to finish selecting benchmarks. The script will also ask the user to specify an experiment name, which the script will use as the directory name in which to store the benchmark results on the client. Finally, the script will ask the user for the number of runs to perform. For each run, the script will run each application benchmark many times, 50 times for most of applications but less times for time-consuming applications. Results for each run will be stored in a numbered subdirectory. We recommend running at least three runs to ensure the reliability of the results. Once the user sets all options, the script will automatically install the benchmarks on both the server (including VMs) and the client, if they are not yet installed, then run the benchmarks. For example, to run the Netperf STREAM benchmark in a running L2 VM on the server using 3 runs and store the results in a directory `L2-dvh` on the client, the script would be run as follows:

```
# cd scripts
# ./run-benchmarks.sh L2
[0] ==== Start Test =====
[1] All
[2] Hackbench
[3] mysql
[4] netperf-rr
[5] netperf-stream
[6] netperf-maerts
[7] apache
[8] memcached
Type test number(Enter 0 to start tests): 5

[0] ==== Start Test =====
[1] All
[2] Hackbench
[3] mysql
[4] netperf-rr
*[5] netperf-stream
[6] netperf-maerts
[7] apache
[8] memcached
Type test number(Enter 0 to start tests): 0
Enter test name: L2-dvh
How many times to repeat? 3
```

A.6 Evaluation and expected result

Once all selected benchmarks finished running on the client, the results can be obtained by running the `results.py` script on the client in the `scripts` directory with a command-line argument that is the test name entered for the `run-benchmarks.sh` script. The script will show the results in a CSV-like format, where each line of results will have one number for each run, and each column represents one complete run. For example, the following shows how to run `results.py` to get the results from the Netperf STREAM example discussed above:

```
# ./results.py L2-dvh
netperf-stream
----------netperf-stream------
9413.81,9413.92,9412.64
9414.22,9413.71,9413.46
... (47 more lines)
9414.13,9414.27,9414.41
---------------------------
```

We determine the performance of each application by choosing the best performance number among the average numbers for each run. The best number is the one that has the highest value for benchmarks measuring transaction rate or data transfer rate such as Netperf, Apache, and Memcached. For benchmarks measuring the elapsed time for a given amount of work such as MySQL and Hackbench, the best number is the one that has the lowest value. For the Netperf STREAM example above, we calculate the average values for three different runs respectively, and choose the highest value among the three average numbers.

We evaluate VM performance by comparing against the native execution of the same workload to quantify performance overhead. For benchmarks choosing the highest value as the best number, the overhead is calculated by dividing the VM performance number by the native execution performance number. For benchmarks choosing the lowest value as the best number, the overhead is calculated the other way around, by dividing the native execution performance number by the VM performance number. The expected overheads are shown in Figure 7 and Figure 9.

References

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

[2] ARM Ltd. 2011. ARM Generic Interrupt Controller Architecture version 2.0 ARM IHI 0048B.

[3] ARM Ltd. 2013. ARM Architecture Reference Manual ARMv8-A DDI0487A.a.

[4] ARM Ltd. 2016. ARM Generic Interrupt Controller Architecture Specification GIC architecture version 3.0 and version 4.0 ARM IHI 0069C.

[5] Jeff Barr. 2017. Now Available – Compute-Intensive C5 Instances for Amazon EC2. AWS News Blog. Retrieved Jan 21, 2020 from https://aws.amazon.com/blogs/aws/now-available-compute-intensive-c5-instances-for-amazon-ec2/

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

[7] Paolo Bonzini. 2018. Migration with directly assigned devices is possible? KVM Mailing List. Retrieved Jan 21, 2020 from https://marc.info/?l=kvm&m=152459004513285&w=2

[8] Edouard Bugnion, Jason Nieh, and Dan Tsafrir. 2017. *Hardware and Software Support for Virtualization.* Morgan and Claypool Publishers.

[9] Cesare Cantu. 2013. Network Interface Card Device Pass-through with Multiple Nested Hypervisors. US Patent US9176767B2.

[10] Sean Christopherson. 2019. KVM: nVMX: Disable intercept for FS/GS base MSRs in vmcs02 when possible. Linux Kernel Source Tree. Retrieved Jan 21, 2020 from https://github.com/torvalds/linux/commit/d69129b4e46a7b61dc956af038d143eb791f22c7

[11] Citrix. 2020. Citrix and AWS partner to enable application elasticity and scale. Retrieved Jan 21, 2020 from https://www.citrix.com/global-partners/amazon-web-services/

[12] CloudShare. 2019. Infrastructure. Retrieved Jan 21, 2020 from https://www.cloudshare.com/technology/nested-virtualization/

[13] Christoffer Dall. 2018. *The Design, Implementation, and Evaluation of the Linux ARM Hypervisor.* Ph.D. Dissertation. Columbia University.

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

[15] Christoffer Dall, Shih-Wei Li, and Jason Nieh. 2017. 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, 221–234.

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

[17] Christoffer Dall and Jason Nieh. 2014. 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, 333–347.

[18] Yaozu Dong, Yu Chen, Zhenhao Pan, Jinquan Dai, and Yunhong Jiang. 2012. ReNIC: Architectural Extension to SR-IOV I/O Virtualization for Efficient Replication. *ACM Transactions on Architecture and Code Optimization (TACO)* 8, 4 (Jan. 2012), 40:1–40:22.

[19] DPDK. 2015. Poll Mode Driver for Emulated Virtio NIC. Retrieved Jan 21, 2020 from https://doc.dpdk.org/guides/nics/virtio.html

[20] DPDK. 2019. Data Plane Development Kit. Retrieved Jan 21, 2020 from https://dpdk.org/

[21] Joy Fan. 2017. Nested Virtualization in Azure. Azure Blog. Retrieved Jan 21, 2020 from https://azure.microsoft.com/en-us/blog/nested-virtualization-in-azure/

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

[23] Google Cloud. 2018. Enabling Nested Virtualization for VM Instances. Retrieved Jan 21, 2020 from https://cloud.google.com/compute/docs/instances/enable-nested-virtualization-vm-instances

[24] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. 2012. 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, 411–422.

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

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

[27] Intel Corporation. 2013. 4th Generation Intel Core vPro Processors with Intel VMCS Shadowing. Retrieved Jan 21, 2020 from http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-vmcs-shadowing-paper.pdf

[28] Rick Jones. 2010. Netperf. Retrieved Jan 21, 2020 from https://github.com/HewlettPackard/netperf

[29] Richard WM Jones. 2014. Super-nested KVM. Retrieved Jan 21, 2020 from https://rwmj.wordpress.com/2014/07/03/super-nested-kvm/

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

[31] KVM. 2018. Tuning KVM — KVM,. Retrieved Jan 21, 2020 from https://www.linux-kvm.org/index.php?title=Tuning_KVM&oldid=173911

[32] Alex Landau, Muli Ben-Yehuda, and Abel Gordon. 2011. SplitX: Split Guest/Hypervisor Execution on Multi-core. In *Proceedings of the 3rd Workshop on I/O Virtualization (WIOV 2011)*. Portland, OR.

[33] Shih-Wei Li, John S. Koh, and Jason Nieh. 2019. 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, 1357–1374.

[34] Wanpeng Li. 2018. KVM: X86: Provide a capability to disable HLT intercepts. Linux Kernel Source Tree. Retrieved Jan 21, 2020 from https://github.com/torvalds/linux/commit/caa057a2cad647fb368a12c8e6c410ac4c28e063

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

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

[37] Microsoft. 2009. Windows XP Mode. Retrieved Jan 21, 2020 from https://www.microsoft.com/en-us/download/details.aspx?id=8002

[38] Microsoft. 2017. Virtualization-based Security (VBS). Retrieved Jan 21, 2020 from https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs

[39] Timothy Prickett Morgan. 2016. Azure Stack Gives Microsoft Leverage Over AWS, Google. The Next Platform. Retrieved Jan 21, 2020 from https://www.nextplatform.com/2016/01/26/azure-stack-gives-microsoft-leverage-over-aws-google/

[40] Oracle. 2019. Oracle Cloud Infrastructure Ravello Service. Retrieved Jan 21, 2020 from https://docs.oracle.com/en/cloud/iaas/ravello-cloud/index.html

[41] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. 2002. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*. Boston, MA, 361–376.

[42] Zhenhao Pan, Yaozu Dong, Yu Chen, Lei Zhang, and Zhijiao Zhang. 2012. 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, 109–120.

[43] Ravello Community. 2016. Nested virtualization: How to run nested KVM on AWS or Google Cloud. Ravello Blog. Retrieved Jan 21, 2020 from https://blogs.oracle.com/ravello/run-nested-kvm-on-aws-google

[44] Robert Ricci, Eric Eide, and The CloudLab Team. 2014. Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *USENIX ;login:* 39, 6 (Dec. 2014), 36–38.

[45] Rusty Russell. 2008. Virtio: Towards a De-facto Standard for Virtual I/O Devices. *ACM SIGOPS Operating Systems Review* 42, 5 (July 2008), 95–103.

[46] Rusty Russell, Yanmin Zhang, Ingo Molnar, and David Sommerseth. 2008. Improve hackbench. Linux Kernel Mailing List. Retrieved Jan 21, 2020 from http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c

[47] Simon Sharwood. 2017. AWS adopts home-brewed KVM as new hypervisor. The Register. Retrieved Jan 21, 2020 from https://www.theregister.co.uk/2017/11/07/aws_writes_new_kvm_based_hypervisor_to_make_its_cloud_go_faster/

[48] Paul Sim. 2013. KVM Performance Optimization. Retrieved Jan 21, 2020 from https://www.slideshare.net/janghoonsim/kvm-performance-optimization-for-ubuntu

[49] SUSE. 2020. Disk Cache Modes. Retrieved Jan 21, 2020 from https://documentation.suse.com/sles/12-SP4/html/SLES-all/cha-cachemodes.html

[50] The Apache Software Foundation. 2015. ab - Apache HTTP server benchmarking tool. Retrieved Jan 21, 2020 from http://httpd.apache.org/docs/2.4/programs/ab.html

[51] Cheng-Chun Tu, Michael Ferdman, Chao-tung Lee, and Tzi-cker Chiueh. 2015. 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, 1–15.

[52] Lluís Vilanova, Nadav Amit, and Yoav Etsion. 2019. Using SMT to Accelerate Nested Virtualization. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA 2019)*. Phoenix, AZ, 750–761.

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

[54] Dan Williams, Yaohui Hu, Umesh Deshpande, Piush K. Sinha, Nilton Bila, Kartik Gopalan, and Hani Jamjoom. 2016. 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, 79–92.

[55] Xen Project wiki. 2014. Network Throughput and Performance Guide. Retrieved Jan 21, 2020 from http://wiki.xen.org/wiki/Network_Throughput_and_Performance_Guide

[56] Xen Project wiki. 2018. Nested Virtualization in Xen. Retrieved Jan 21, 2020 from https://wiki.xenproject.org/wiki/Nested_Virtualization_in_Xen

[57] Xen Project wiki. 2019. Xen PCI Passthrough. Retrieved Jan 21, 2020 from https://wiki.xen.org/wiki/Xen_PCI_Passthrough

[58] Xin Xu and Bhavesh Davda. 2017. A Hypervisor Approach to Enable Live Migration with Passthrough SR-IOV Network Devices. *ACM SIGOPS Operating Systems Review* 51, 1 (Sept. 2017), 15–23.

[59] Edwin Zhai, Gregory D. Cummings, and Yaozu Dong. 2008. Live Migration with Pass-through Device for Linux VM. In *Proceedings of the 2008 Ottawa Linux Symposium (OLS 2008)*. Ottawa, ON, Canada, 261–267.