NEVE: Nested Virtualization Extensions for ARM

Jin Tack Lim[1], Christoffer Dall[1], Shih-Wei Li[1], Jason Nieh[1], Marc Zyngier[2]
[1]Columbia University [2]ARM Ltd
{jintack ,cdall, shihwei, nieh}@cs.columbia.edu, marc.zyngier@arm.com

ABSTRACT

Nested virtualization, the ability to run a virtual machine inside another virtual machine, is increasingly important because of the need to deploy virtual machines running software stacks on top of virtualized cloud infrastructure. As ARM servers make inroads in cloud infrastructure deployments, supporting nested virtualization on ARM is a key requirement, which has been met recently with the introduction of nested virtualization support to the ARM architecture. We build the first hypervisor to use ARM nested virtualization support and show that despite similarities between ARM and x86 nested virtualization support, performance on ARM is much worse than on x86. This is due to excessive traps to the hypervisor caused by differences in non-nested virtualization support. To address this problem, we introduce a novel paravirtualization technique to rapidly prototype architectural changes for virtualization and evaluate their performance impact using existing hardware. Using this technique, we propose Nested Virtualization Extensions for ARM (NEVE), a set of simple architectural changes to ARM that can be used by software to coalesce and defer traps by logging the results of hypervisor instructions until the results are actually needed by the hypervisor or virtual machines. We show that NEVE allows hypervisors running real application workloads to provide an order of magnitude better performance than current ARM nested virtualization support and up to three times less overhead than x86 nested virtualization. NEVE will be included in ARMv8.4, the next version of the ARM architecture.

SOSP '17, October 28, 2017, Shanghai, China

ACM ISBN 978-1-4503-5085-3/17/10…$15.00
https://doi.org/10.1145/3132747.3132754

CCS CONCEPTS

• **Computer systems organization** → **Architectures**; • **Software and its engineering** → **Virtual machines**; **Operating systems**; • **Computing methodologies** → *Simulation evaluation*;

KEYWORDS

Nested virtualization, hypervisors, performance, ARM

ACM Reference Format:
Jin Tack Lim, Christoffer Dall, Shih-Wei Li, Jason Nieh, and Marc Zyngier. 2017. NEVE: Nested Virtualization Extensions for ARM. In *Proceedings of SOSP '17, Shanghai, China, October 28, 2017,* 17 pages.
https://doi.org/10.1145/3132747.3132754

1 INTRODUCTION

Nested virtualization is the discipline of running virtual machines (VMs) inside virtual machines. In other words, to nest hypervisors is to run multiple levels of hypervisors. Nested virtualization is increasingly important as new use cases for virtualization are on the rise. For example, operating systems (OSes) including Microsoft Windows now have built-in hypervisors to support legacy applications and require nested virtualization support to run in VMs. Similarly, deploying VMs on top of Infrastructure-as-a-Service (IaaS) cloud providers is becoming more commonplace and requires nested virtualization support [12, 14, 22, 43].

While the x86 architecture has dominated the server and cloud infrastructure markets, the ARM architecture is leveraging its dominance in the mobile and embedded space to make inroads in cloud infrastructure deployments [19]. Because of the demand for nested virtualization in these markets, architecture support for nested virtualization has recently been added in the latest ARMv8.3 architecture [11]. However, no ARMv8.3 hardware exists yet and, as a consequence, no hypervisors have been developed for ARM that support nested virtualization. While nested virtualization can deliver reasonable performance on x86 [10], it remains an unexplored technology on ARM. Given the growing popularity of virtualization on ARM and attractive use cases for nesting, investigating the future for nesting support on ARM is important.

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

Using this approach, we build the first ARM hypervisor to support nested virtualization. We modified KVM/ARM [18] to support ARMv8.3 nested virtualization features. Both the hypervisor design and ARMv8.3 are based on a trap-and-emulate approach similar to how software supports nested virtualization on x86. Despite these similarities, we show that ARMv8.3 nested virtualization performance is quite poor and significantly worse than x86. Our results provide the first quantitative comparison between ARM and x86 nested virtualization performance, and provide crucial insight regarding virtualization support on other emerging architectures. We identify for the first time how differences in the design of single-level hardware virtualization support, which do not cause significant performance impact for non-nested virtualization, end up causing a very significant performance impact for nested virtualization due to excessive traps to the hypervisor.

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

Using our paravirtualization approach for architecture performance evaluation, we have built a complete hypervisor for nested virtualization by modifying KVM/ARM to use NEVE on existing hardware. Our measurements on real application workloads show that NEVE can provide up to an order of magnitude better performance than the latest ARMv8.3 architecture, and up to three times less overhead than x86 nested virtualization. NEVE will be included in the next ARM architecture, ARMv8.4.

2 BACKGROUND

Nested virtualization is the ability to run multiple levels of VMs. Normal single-level virtualization runs a hypervisor on the hardware and creates a virtual machine environment similar to the underlying hardware for the VM. This allows a standard OS designed to run on the underlying hardware to run without modifications inside the virtual machine environment. With multiple levels of 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 next level hypervisor. With more levels of virtualization, we also refer to the host hypervisor as the L0 hypervisor, 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, etc. While classically virtualizable architectures are also recursively virtualizable [35], non-virtualizable architectures like x86 and ARM which have specific hardware features to support running hypervisors may not necessarily be recursively virtualizable.

The ARMv8 architecture [7] includes the ARM Virtualization Extensions (VE). VE adds a more privileged CPU mode, known as an exception level, called EL2. ARM CPU exception levels EL0, EL1, and EL2 are designed to run user applications, an OS kernel, and a hypervisor, respectively. Each exception level has different sets of system registers, which are only accessible from the same or more privileged exception level. For single-level trap-and-emulate virtualization, VMs are executed in EL0 and EL1. CPU virtualization works by letting software executing in EL2 configure the CPU to trap to EL2 on events and instructions that cannot be safely executed by a VM, for example on hardware interrupts and I/O instructions. Memory virtualization works by allowing software in EL2 to point to a set of page tables, Stage-2 page tables, used to translate the VM's view of physical addresses to machine addresses, while Stage-1 page tables can be used and managed by the VM without trapping to the hypervisor to translate virtual to physical addresses. Interrupt

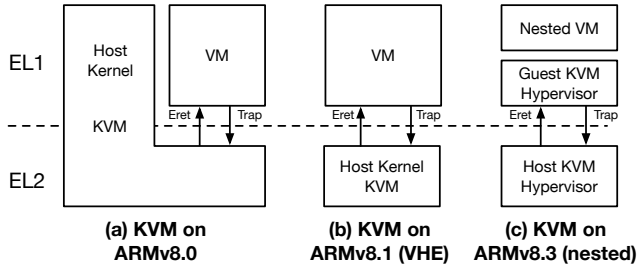

Figure 1: ARM Hardware Virtualization Extensions

virtualization works by allowing the hypervisor to inject virtual interrupts to VMs, which VMs can acknowledge and complete without trapping to the hypervisor.

ARMv8.1 introduced the Virtualization Host Extensions (VHE) [15]. Without VHE, hosted hypervisors which are integrated with an OS kernel, needed to split their OS and hypervisor functionality across EL1 and EL2, respectively [18] as shown in Figure 1(a). VHE allows running both the OS kernel and hypervisor functionality in EL2 as shown in Figure 1(b). VHE expands the capabilities of EL2 so that it can be functionally equivalent to EL1, including adding additional EL2 system registers. VHE supports running existing OS kernels written for EL1 in EL2 without having to modify the OS source code. VHE transparently redirects EL1 system register access instructions to access EL2 system registers instead. New instructions are added for the hypervisor to access the EL1 system registers which belong to the VM context.

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

A better alternative is running the guest hypervisor in EL1. Unfortunately, this does not work without ARMv8.3 nested virtualization support. Hypervisor instructions do not trap to EL2 when executed in EL1, but cause exceptions directly to the guest hypervisor in EL1. This would typically lead to an unmodified hypervisor crashing if executed in EL1. For example, suppose the guest hypervisor wishes to configure its own page table base register. Since this EL2 register is accessed using a hypervisor instruction which does not trap to EL2 but instead causes an exception in EL1, attempts to change the register would cause an unexpected exception to the guest hypervisor executing in EL1, likely leading to a software crash. To address this limitation, ARM recently introduced nested virtualization support in the ARMv8.3 revision of the architecture [11]. It works in three parts. First, it enables trapping of hypervisor instructions executed in EL1 to EL2. Second, it disguises the deprivileged execution by telling the guest hypervisor that it runs in EL2 if it reads the CurrentEL register, which contains the current exception level. Third, it supports using the EL2 page table format in EL1. The resulting configuration using KVM on ARMv8.3 is shown in Figure 1(c).

Comparison to x86 ARMv8.3 nested virtualization support is similar to x86 in that guest hypervisor instructions can be configured to trap to the host hypervisor. However, the core hardware virtualization support is different. We limit our discussion of x86 to Intel VT as it is similar to AMD-V for all purposes discussed here. While ARM VE provides a separate CPU privilege level, EL2, with its own set of features and register state, Intel VT provides root vs. non-root mode, completely orthogonal to the CPU privilege levels, each of which supports the same full range of user and kernel mode functionality. Both ARM and Intel trap into their respective EL2 and root modes, but transitions between root and non-root mode on Intel are implemented with a VM Control Structure (VMCS) residing in normal memory, to and from which hardware state is automatically saved and restored when switching to and from root mode, for example when the hardware traps from a VM to the hypervisor. ARM instead has a simpler hardware mechanism to transition between EL1 and EL2, but leaves it up to software to decide what state needs to be saved and restored, providing more flexibility to optimize what is done for each transition.

Because of these differences in the core hardware virtualization support, ARMv8.3 must provide some additional mechanisms not necessary for x86 to provide the same level of support for nested virtualization. First, since ARM augments the existing CPU privilege level for virtualization support, as opposed to introducing an orthogonal mechanism, ARMv8.3 needs to disguise the CPU privilege level so that a hypervisor that normally runs in EL2 does not know that it is running in EL1 as a guest hypervisor. Second, because EL2 is a separate privilege level with its own page table format that differs from the EL1 page table format, ARMv8.3 allows a hypervisor, which would normally use the EL2 page table format when run in EL2, to use the same format when run as a guest hypervisor in EL1.

3 PARAVIRTUALIZATION FOR ARCHITECTURE EVALUATION

Unfortunately, ARMv8.3 hardware is not available, and the newest publicly available ARM hardware is still v8.0. As architectural support for virtualization is increasingly common, understanding the performance of these features is important, ideally before they become set in production hardware. However, evaluating new architecture features for virtualization is challenging because of costs associated with prototyping new hardware and the need to understand the interaction of both hardware and software. Chip vendors use cycle-accurate simulators to measure performance, but they are typically many orders of magnitude slower than real hardware, making it hard to evaluate real-life workloads. Booting a full virtualization stack including the hypervisor and VM can take days, and even then, measuring key application performance characteristics such as fast I/O performance using 10G Ethernet is still not possible. Furthermore, simulators of commercial architecture designs are themselves quite complex to build and often closed and proprietary, limiting their availability in practice. Software developers often can only use simpler architecture models before hardware is available, at the cost of not being able to measure any real architecture performance.

To overcome this challenge, we introduce an existing idea, paravirtualization, in a new context. Paravirtualization allows for a software interface to a VM that differs slightly from the underlying hardware [46]. It is used to make hypervisors simpler and faster by avoiding certain architecture features that are complex or difficult to virtualize efficiently. We instead use paravirtualization to allow us to build hypervisors using new architecture features that do not exist on current hardware, and measure the performance of a full virtualization stack using new architecture features at native execution speeds on existing hardware.

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

For example, as discussed in Section 2, current ARM server hardware does not support nested virtualization, because when a hypervisor runs inside a VM on top of another hypervisor, various instructions that it executes do not trap to the underlying hypervisor for proper execution, but instead simply fail improperly. However, if we replace those hypervisor instructions with instructions that do trap on current hardware and the trap cost is expected to remain similar in future hardware, we can obtain similar relative performance to future hardware that supports nested virtualization with correct trapping behavior.

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

Second, the approach assumes that certain types of traps are interchangeable in terms of performance. For example, on ARM, the trap cost using an explicit trap instruction should be similar to the cost of any system register access instruction that traps. Only the cost of the trap itself needs to remain similar; the overall cost of handling the respective trap can be quite different. This assumption is likely to be true in most cases and we have validated it on ARM hardware, as discussed in Section 5.

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

4 KVM/ARM NESTED VIRTUALIZATION FOR ARMV8.3

Because ARMv8.3 hardware is not yet available, we leverage our paravirtualization approach discussed in Section 3 to allow us to design, implement, and evaluate the first ARM hypervisor to support nested virtualization using ARMv8.3 architectural support on existing ARMv8.0 hardware. Since both ARM and x86 provide a single level of architectural virtualization support, we take an approach similar to Turtles [10] for supporting nested virtualization on x86, where multiple levels of virtualization are multiplexed onto the single level of architectural support available. We have implemented nested virtualization support on ARM by modifying KVM/ARM [18], the widely-used mainline Linux ARM hypervisor. There are two kinds of modifications: (1) changes to KVM/ARM as a host hypervisor to support running guest

hypervisors, and (2) paravirtualization of KVM/ARM to run as a guest hypervisor on ARMv8.0 with similar behavior as an unmodified KVM/ARM guest hypervisor on ARMv8.3.

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

To mimic ARMv8.3 behavior using ARMv8.0 hardware so that hypervisor instructions run by the guest hypervisor trap as needed to the host hypervisor, we paravirtualize the guest hypervisor by replacing the hypervisor instructions with `hvc` instructions. An `hvc` instruction takes a 16-bit operand and generates an exception to EL2, which can read the 16-bit operand back from a system register. We encode the hypervisor instructions using the 16-bit operand, so that on the trap to EL2, the host hypervisor is informed of the original guest hypervisor instruction that was replaced by an `hvc` and can emulate the behavior of that instruction.

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

There are four kinds of hypervisor instructions that are paravirtualized to mimic ARMv8.3 behavior so they trap if executed by the guest hypervisor on ARMv8.0 hardware. First, instructions that can only run in EL2, such as those that directly access EL2 registers, are undefined when executed in EL1 on ARMv8.0, so they are paravirtualized to trap to EL2 to access virtual EL2 state.

Second, instructions that run as part of the hypervisor and access EL1 registers are paravirtualized to trap to EL2 because they will now interfere with the execution of the guest hypervisor which is really running in EL1. For example, an ARM hypervisor will configure EL1 registers to run a VM with its guest OS in EL1. This works fine if the hypervisor is running in EL2 and writes to EL1 registers for the VM, but is problematic if the hypervisor is deprivileged running in EL1, because it will then unknowingly be overwriting its own EL1 register state. Instead, these EL1 access instructions must trap to the host hypervisor which will then emulate the instruction on virtual EL1 register state. The host hypervisor is then responsible for multiplexing EL1 state between the guest hypervisor and the nested VM by context switching the hardware EL1 state to the nested VM's virtual EL1 state when the nested VM runs. For some EL1 access instructions, existing ARMv8.0 mechanisms are used by the host hypervisor to configure them to trap, avoiding paravirtualization of these instructions.

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

Finally, VHE adds a number of new instructions that are undefined on ARMv8.0 which must be paravirtualized to trap to EL2 so they can be emulated. These new instructions are used to access EL1 state when running in EL2 with register access redirection enabled, as explained in Section 2. Because these instructions are not defined on ARMv8.0, they generate an exception to EL1 when executed by a guest hypervisor, instead of trapping to EL2. To allow guest hypervisors to be configured with VHE on ARMv8.0, these instructions are paravirtualized to trap as they would on ARMv8.3. Because VHE is designed to make EL2 work the same way as EL1 and because the guest hypervisor already runs in EL1, running a VHE guest hypervisor works trivially without further changes.

Memory virtualization is done using shadow page tables [1] to handle additional levels of memory translation imposed by nested virtualization. ARM hardware supports only two stages of address translation via Stage-1 and Stage-2 page tables. Nested virtualization requires at least three: L2 VM virtual address (VA) to L2 VM physical address (PA),

L2 VM PA to L1 VM PA, L1 VM PA to L0 PA. Similar to previous work [10], the host hypervisor creates shadow Stage-2 page tables to map from L2 VM PAs to L0 PAs by collapsing Stage-2 page tables from the guest and host hypervisors. The Stage-1 MMU translates L2 VAs to L2 PAs using the L2 guest OS's page tables, and the Stage-2 MMU then translates L2 VM PAs to L0 PAs using the shadow page tables.

Interrupt virtualization is accomplished by providing a hypervisor control interface to a guest hypervisor via trap-and-emulate. This interface is used by a hypervisor to control virtual interrupts for higher level VMs and is multiplexed onto the single-level ARM virtual interrupt support in the ARM Generic Interrupt Controller (GIC). When a guest hypervisor programs registers in the hypervisor control interface, this must trap to the host hypervisor to sanitize and translate the payload before writing shadow copies of the register payload into the hardware control interface. The hypervisor control interface is memory mapped with GICv2 and therefore trivially traps to EL2 when not mapped in the Stage-2 page tables, but GICv3 uses system registers and must use paravirtualization of the guest hypervisor to mimic ARMv8.3's behavior of trapping EL1 accesses to EL2 on ARMv8.0.

5 EVALUATION OF ARMV8.3 NESTED VIRTUALIZATION

We present some experimental results that quantify the nested virtualization performance of ARMv8.3 based on running our paravirtualized KVM/ARM guest hypervisor on our KVM/ARM host hypervisor on multicore ARM hardware. We also measure the performance of a KVM x86 guest hypervisor on top of a KVM x86 host hypervisor to compare against a more mature nested virtualization solution with a similar hypervisor design; KVM x86 is based on Turtles. These results provide the first measurements of ARM nested virtualization as well as the first comparison of nested virtualization between ARM and x86. Experiments were conducted using server hardware in CloudLab [37].

ARM measurements were done using HP Moonshot m400 servers, each with a 64-bit ARMv8-A 2.4 GHz Applied Micro Atlas SoC with 8 physical CPU cores. Each m400 node had 64 GB of RAM, a 120 GB SATA3 SSD for storage, and a Dual-port Mellanox ConnectX-3 10 GbE NIC. x86 measurements were done using Cisco UCS SFF 220 M4 servers, each with two Intel E5-2630 v3 8-core 2.4 GHz CPUs. Hyperthreading was disabled on the nodes to provide a similar hardware configuration to the ARM servers. Each node has 128 GB of ECC memory (8x16 GB DDR4 1866 MHz dual rank RDIMMs), a 2x1.2 TB 10K RPM 6G SAS SFF HDD for storage, and a Dual-port Cisco UCS VIC1227 VIC MLOM 10 GbE NIC. The x86 hardware includes VMCS Shadowing [27], the latest

	ARMv8.3			x86	
Micro-benchmark	**VM**	**Nested VM**	**Nested VM VHE**	**VM**	**Nested VM**
Hypercall	2,729	422,720	307,363	1,188	36,345
Device I/O	3,534	436,924	312,148	2,307	39,108
Virtual IPI	8,364	611,686	494,765	2,751	45,360
Virtual EOI	71	71	71	316	316

Table 1: Microbenchmark Cycle Counts

x86 hardware support for nested virtualization. All servers were connected via 10 GbE, and the interconnecting network switch easily handles multiple sets of nodes communicating with full 10 Gb bandwidth.

To provide comparable measurements, we kept the software environments across all hardware platforms and hypervisors the same as much as possible. For the host and guest hypervisors, we used KVM in Linux 4.10.0-rc3 with QEMU 2.3.50, with our modifications for ARM nested virtualization. KVM/ARM can be configured to run with or without VHE support; we ran experiments with both versions as guest hypervisors. KVM was configured with its standard VHOST virtio network, and with `cache=none` for virtual block storage devices [25, 32, 42]. All hosts and VMs used Ubuntu 14.04 with the same Linux 4.10.0-rc3 kernel and software configuration. All VMs used paravirtualized I/O using virtio-net and virtio-block over PCI.

We ran experiments in two configurations, in a VM (no nesting) and in a nested VM. The VM was configured with 4 cores and 12 GB RAM running on KVM with 8 cores and 16 GB RAM. The nested VM was configured with 4 cores and 12 GB RAM running on a KVM guest hypervisor with 6 cores and 16 GB RAM running on the host KVM hypervisor with 8 cores and 20 GB RAM. The CPU and memory configurations were selected to provide the same hardware resources to the VM or nested VM used for running the experiments while ensuring more than adequate hardware resources for the underlying hypervisor(s).

We leveraged the kvm-unit-test microbenchmarks [33] to quantify important micro-level interactions between the hypervisor and its VM. Table 1 shows the results for running kvm-unit-test in the VM and nested VM configurations for ARMv8.3, with and without VHE, and x86. Measurements are shown in cycles instead of time to provide a useful comparison across hardware. Despite using a similar hypervisor architecture on ARM and x86, which both leverage trap-and-emulate hardware support for nested virtualization, as well as sharing the same architecture-independent parts of the KVM implementation, the measurements show that ARMv8.3 has drastically worse nested virtualization performance than x86.

The Hypercall benchmark measures the cost of switching from a VM to the hypervisor, and immediately back to the

VM without doing any work in the hypervisor. Compared to using a VM, making hypercalls from a nested VM to a guest hypervisor on ARMv8.3 is 155 and 113 times more expensive using a non-VHE and VHE guest hypervisor, respectively. When a nested VM makes a hypercall, it first traps to the host hypervisor running in EL2. The host hypervisor then forwards this hypercall to the guest hypervisor by emulating an exception to the virtual EL2 mode in the VM. When the guest hypervisor processes the hypercall, it simply returns back to the nested VM. However, the process of transitioning between the guest hypervisor and the nested VM involves executing many hypervisor instructions that trap to the host hypervisor, which ends up being very expensive.

The Device I/O benchmark measures the cost of accessing an emulated device in the hypervisor. This is a frequent operation for many device drivers and provides a common baseline for accessing I/O devices emulated in the hypervisor. Device I/O is more costly than Hypercall because it emulates the device in addition to performing similar operations to Hypercall. This additional work reduces the relative overhead of running in a nested VM versus a VM, but the overhead is still hundreds of thousands of cycles on ARMv8.3 compared to tens of thousands of cycles on x86.

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

The Virtual EOI benchmark measures the cost of completing a virtual interrupt, also known as End-Of-Interrupt. The interrupt controllers of both platforms, GIC [6] on ARM and APICv [28] on x86, include support for completing interrupts directly in the VM without trapping to the hypervisor. As a result, this operation is much less expensive than the other benchmarks which trap. The KVM host hypervisor provides support on both ARM and x86 so that nested VMs can use hardware-accelerated virtual interrupt completion, resulting in the same cost for both VMs and nested VMs.

In all cases except Virtual EOI, the cost of running the microbenchmarks in a nested VM on ARMv8.3 is prohibitively expensive compared to running in a VM. Compared to x86, nested VM performance on ARMv8.3 imposes more than an order of magnitude more overhead in terms of cycle counts, and up to 7 times more overhead in terms of relative performance compared to a VM. While trap-and-emulate nested virtualization provides reasonable performance on x86, it does not on ARMv8.3.

To investigate the reasons behind the poor ARMv8.3 performance, we measured the average number of traps to the host hypervisor when running the Hypercall benchmark. While Hypercall only causes a single trap when running in a VM, it causes 126 and 82 traps to the host hypervisor when running in a nested VM using a non-VHE and VHE guest hypervisor, respectively. Clearly, each trap, also known as an exit, from the nested VM results in a multitude of additional traps from the guest hypervisor to the host hypervisor. This is a major source of overhead for nested virtualization and is called the exit multiplication problem [10].

The guest hypervisor using VHE performs better than without VHE, because it traps less often. When KVM/ARM runs with VHE enabled, it uses EL1 system register access instructions wherever possible with the expectation that the hardware redirects these instructions to EL2 registers, as discussed in Section 2. When this is done as a VHE guest hypervisor running in EL1 on ARMv8.0 hardware, it simply accesses EL1 registers directly without trapping to the host hypervisor, and the host hypervisor configures the EL1 hardware registers with the guest hypervisor's state. In contrast, a non-VHE guest hypervisor can only access EL2 state using EL2 system register access instructions, and each such access will trap to the host hypervisor since EL2 registers are not accessible at EL1. Despite this reduction in the number of traps for a VHE guest hypervisor, its nested virtualization performance remains poor.

Our measurements of ARMv8.3 nested virtualization performance are based on replacing guest hypervisor instructions on ARMv8.0 that do not trap as they would on ARMv8.3 with `hvc` instructions, which are explicit trap instructions, to mimic ARMv8.3 behavior. The replaced instructions are mostly system register access instructions along with a few `eret` instructions. On ARM, the cost of a trap should be evaluated in two parts: (1) finding out that you need to generate an exception, and (2) generating the exception. The first can range from expensive (memory fault) to being free (`hvc` instruction), with a system register trap being almost free. The second is a fixed cost for all instructions. As a result, the cost of traps for the replaced instructions is expected to be very similar to that of an `hvc` instruction on all implementations of the ARM architecture.

We further measured the trap cost of several different system register access instructions that trap on current ARMv8.0 hardware and compared their cost with an `hvc` instruction. In all cases, trapping from EL1 to EL2 was between 68 to 76 cycles, and returning from a trap to EL2 back to EL1 was

65 cycles. The difference in trap costs across different instructions was less than 10% overall and less than 10 cycles. These measurements on current ARM hardware support our assumption that hvc instructions can be used as a suitable replacement to mimic ARMv8.3 instructions that trap on system register accesses with similar performance.

6 NEVE: NESTED VIRTUALIZATION EXTENSIONS

Nested virtualization support as introduced in ARMv8.3 traps hypervisor instructions from a deprivileged guest hypervisor running in EL1 to a host hypervisor running in EL2. A single exit from a nested VM can result in the guest hypervisor issuing many hypervisor instructions, resulting in a multitude of additional traps from the guest hypervisor to the host hypervisor. Many hypervisor instructions need to trap because they access system registers. If we can reduce the number of accesses to system registers that need to trap, we can potentially reduce overhead and improve the performance of nested virtualization on ARM.

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

Based on this observation, we propose NEVE, an addition to the ARMv8.3 architecture that avoids traps from the guest hypervisor to the host hypervisor for a wide range of hypervisor instructions that access system registers. NEVE supports unmodified guest hypervisors, both hosted and standalone designs, and unmodified guest OSes. NEVE has three key mechanisms. First, it avoids traps to the host hypervisor for VM registers and instead adds hardware support to store VM registers in memory until they are actually needed for VM execution. In ARMv8.3, when a guest hypervisor accesses a VM system register, it traps to the host hypervisor, which simply stores this value in memory in a software-managed data structure, and later programs this value into physical registers when running the nested VM. NEVE instead supports this operation in hardware by using an architecturally defined storage format and transparently rewriting system register access instructions into normal memory accesses.

Second, NEVE reduces traps to the host hypervisor for hypervisor control registers by instead identifying and using equivalent registers that can be accessed without trapping. In ARMv8.3, when the guest hypervisor writes to a hypervisor control register and traps to the host hypervisor, in many cases, the host hypervisor handles the trap by writing into an equivalent EL1 register. For example, the guest hypervisor will write the base address of the exception vector for itself in VBAR_EL2 which will trap to the host hypervisor, which in turn needs to write the address to VBAR_EL1, the equivalent EL1 register, so that the guest hypervisor running in EL1 will handle exceptions correctly. In cases where the EL1 and EL2 registers have the same format, NEVE instead supports this operation in hardware by transparently redirecting accesses to EL2 registers to EL1 registers without trapping to the host hypervisor.

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

6.1 Architecture Specification

NEVE introduces an EL2 Virtual Nested Control Register (VNCR_EL2) which is managed exclusively by the host hypervisor. The host hypervisor can use the VNCR_EL2 to enable and disable NEVE and to configure a *deferred access page* in memory used to store the values of VM system registers. Table 2 shows the bit fields in the VNCR_EL2 register. The BADDR field contains the physical base address of the deferred access page. The layout of the deferred access page can be arbitrarily defined as long as each VM system register is stored at a well-defined offset from BADDR. The Enable bit completely enables or disables NEVE. When the Enable field is set to 1, and the ARMv8.3 nested virtualization support is enabled, all accesses to the VM system registers which would otherwise trap to the host hypervisor are redirected to memory accesses to the deferred access page. Similarly, the register redirection described above for hypervisor control registers is enabled and disabled using the Enable field in the VNCR_EL2.

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

Table 2: VNCR_EL2 Register Fields

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

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

Table 3: VM System Registers

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

Table 3 lists the 27 VM system registers we identified as part of the ARMv8.3 specification which do not affect execution of the hypervisor directly. When enabled, NEVE redirects accesses to these registers to the deferred access page. The VM Trap Control registers control when certain operations performed by the VM trap to the hypervisor and other virtualization features such as Stage-2 translation and virtual interrupts. The VM Execution Control registers are system registers that belong to the VM itself and do not affect hypervisor execution. The Thread ID register, TPIDR_EL2, is

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

Table 4: Hypervisor Control Registers

commonly used by hypervisors to store thread-specific data, but does not affect the hypervisor's execution.

We distinguish two types of hypervisor control registers, normal system registers and GIC registers related to the hypervisor control interface used for interrupt virtualization, discussed in Section 4. When the guest hypervisor executes in virtual EL2, which really runs in EL1, accesses to these EL2 registers would normally trap to the host hypervisor, but NEVE uses two techniques to avoid traps, register redirection and cached copies. Table 4 shows the 17 normal system registers we identified that affect the hypervisor's execution in EL2, and the techniques NEVE used to avoid traps.

Register redirection transparently redirects accesses from an EL2 register to its corresponding EL1 register if it exists and has the same format as the EL2 register. Since the guest hypervisor is really running in EL1, EL2 register accesses can be redirected to corresponding EL1 registers such that changes to the registers have the same impact on the hypervisor's execution when running deprivileged in EL1 as running in EL2 on real hardware. NEVE provides register redirection for 12 EL2 registers with corresponding EL1 registers as shown in Table 4, two of which are grouped separately (VHE) as they were added as part of VHE and are only relevant for VHE hypervisors.

Cached copies (shown as "Trap on write" in Table 4) transparently changes reads from EL2 registers that don't have an equivalent EL1 to instead read a cached copy from the deferred access page. The host hypervisor copies the value of the virtual EL2 register to the deferred access page when running the guest hypervisor to cache the latest value of the register for reads from the guest hypervisor. Writes to these

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

Table 5: Hypervisor Control GIC Registers

registers will trap, allowing the host hypervisor to update the content of the deferred access page as needed. Cached copies are used for four EL2 registers, two of which have similar EL1 registers but with different formats and thus cannot be used with register redirection from EL2 to EL1 registers, namely CNTHCTL_EL2 and CPTR_EL2.

Table 4 lists two EL2 registers, TCR_EL2 and TTBR0_EL2, that may be redirected to corresponding EL1 registers for VHE guest hypervisors only. VHE changes the format of these EL2 registers to be identical to the corresponding EL1 registers. VHE guest hypervisors can therefore access these registers directly using EL1 access instructions. A non-VHE guest hypervisor, however, would use the EL2 register formats, which are incompatible with the EL1 registers, and therefore the EL2 register accesses cannot be redirected to EL1 registers but must instead be supported using cached copies, trapping on writes to these registers.

Table 5 shows the GIC registers in the hypervisor control interface registers we identified that affect the hypervisor's execution in EL2. NEVE uses cached copies in the deferred access page for all of these registers to avoid traps.

ARM also provides performance monitoring, debugging, and timer system registers. We note that accesses to the PMUSERENR_EL0 and PMSELR_EL0 performance monitor control registers can be redirected to the deferred access page like VM system registers, reads from the MDSCR_EL1 debug control register can be redirected to a cached copy so that only writes must trap, and all accesses to the virtual and physical hypervisor timer EL2 registers trap as reads must access the registers directly to obtain correct values updated by hardware. Further details are omitted due to space constraints.

6.2 Recursive Virtualization

NEVE supports multiple levels of nesting, also known as recursive nesting. As discussed in Section 4, recursive nesting is supported with ARMv8.3, because the host hypervisor emulates the same virtual execution environment as the underlying machine including the hardware virtualization support and nesting support. NEVE can further improve the performance of each level of hypervisor. The L0 host hypervisor can create a VM with support for NEVE, which the guest hypervisor will use when running the L2 guest hypervisor. When the L1 guest hypervisor configures NEVE by accessing the VNCR_EL2, we cache the register state to the deferred access page. Because the VNCR_EL2 of the L1 guest hypervisor does not affect the execution of L1 hypervisor, but only affects the execution of the L2 guest hypervisor. On entry to the L2 VM's virtual EL2, the L0 host hypervisor can emulate the behavior of NEVE by using the hardware features directly. This works by translating the VM physical address written by the L1 guest hypervisor into a machine physical address and using this address in the hardware VNCR_EL2. This allows transparently changing register accesses performed by the L2 guest hypervisor into memory and EL1 register accesses. The memory used is provided by the L1 guest hypervisor which can therefore directly access the content of the deferred access page used to support the L2 guest hypervisor running NEVE. In this scenario, NEVE avoids the same amount of traps between the L2 and L1 guest hypervisors as in the normal nested case described above.

6.3 Architectural Impact

NEVE represents a relatively small architectural change. It requires adding the VNCR_EL2 register and adding logic to redirect system register access instructions from VM registers to memory at a specified offset, when NEVE is enabled in the VNCR_EL2 register. It also requires adding logic to redirect instructions accessing EL2 registers to corresponding EL1 registers or to memory on read accesses, when NEVE is enabled. Since ARMv8.1 already supports redirecting system register access instructions to other system registers depending on a run-time configuration, the most invasive part of our proposal is to redirect a system register access to a memory access. To simplify the logic to handle this, we propose that the architecture mandates that the host hypervisor software programs a page-aligned physical address in the VNCR_EL2.BADDR field to avoid the need to perform alignment checks or handle address translation faults.

6.4 Implementation

Although NEVE is designed to work with unmodified guest hypervisors, it requires modest hardware changes to do so. To show how NEVE can be used in the absence of a hardware implementation of NEVE, we describe how we can modify KVM/ARM to use this feature via our paravirtualization approach from Section 3. We can use the same KVM/ARM design from Section 4, but with modifications to CPU virtualization to use NEVE. To implement the deferred access page, we establish a shared memory region between the host and guest hypervisor. We modify KVM/ARM to run as a guest

hypervisor using NEVE by replacing instructions that access VM registers with normal load and store instructions that access the shared memory region. We also modify KVM/ARM to run as a guest hypervisor by replacing instructions that access EL2 hypervisor control registers with instructions that access corresponding EL1 registers to provide the equivalent register redirection functionality shown in Table 4. The resulting guest hypervisor eliminates the same traps to the host hypervisor and provides the same performance characteristics as a hardware system with NEVE.

We run KVM/ARM in two configurations as the guest hypervisor, non-VHE and VHE. Non-VHE KVM/ARM issues EL1 system register access instructions to access EL1 VM system registers and EL2 system register access instructions to access EL2 VM system registers. These are replaced with load and store instructions to mimic NEVE. As described in Section 2, a VHE hypervisor takes advantage of the VHE register redirection feature to allow its integrated OS written for EL1 to run in EL2 without modification. With VHE, EL1 system register access instructions are redirected to EL2 system registers, and KVM/ARM with VHE uses EL1 system register access instructions wherever possible to access EL2 registers, as discussed in Section 5. VHE KVM/ARM running as the guest hypervisor will therefore access its own virtual EL2 register state directly using EL1 system register instructions, and there is no need to replace any of these instructions. However, VHE introduces separate EL12 system register access instructions to access EL1 VM system registers, which are replaced with load and store instructions to mimic NEVE.

6.5 Performance Impact

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

First, consider a legacy KVM/ARM implementation without support for VHE [18]. KVM/ARM saves and restores all the VM system registers and modifies VM trap control registers on every VM exit because it uses the same EL1 hardware state to run the Linux kernel portion of the hypervisor. Furthermore, a non-VHE hosted hypervisor frequently accesses the hypervisor control registers when moving between EL1 and EL2. Each of these register accesses from the guest hypervisor traps, resulting in significant exit multiplication using ARMv8.3, and NEVE provides a significant performance gain for this hypervisor design as shown in Section 7.

Second, consider KVM/ARM in the context of the Virtualization Host Extensions (VHE) [15], which were introduced in ARMv8.1. While KVM/ARM was originally designed to run across both EL1 and EL2, VHE allows the KVM/ARM hypervisor to run entirely in EL2. As a result, KVM/ARM no longer needs to use EL1 system registers and the hypervisor is unaffected by VM trap controls. Therefore, switching between the VM and a VHE hypervisor no longer requires saving and restoring the full VM system register state or configuring VM trap-control registers. However, even with VHE, the current KVM/ARM implementation frequently accesses the VM system registers. The reason is that KVM/ARM saves the VM EL1 context and modifies the VM trap-control registers when switching from the VM to the hypervisor and back, because avoiding these operations while preserving backwards compatibility with non-VHE systems is difficult and would complicate the code base. Furthermore, saving and restoring the full EL1 system register state is still needed when switching between VMs. Therefore, KVM/ARM and similar VHE-enabled hypervisors will benefit from NEVE as shown in Section 7.

Third, consider Xen [48] which runs only in EL2 as a standalone hypervisor. Since Xen does not need to use the VM system registers for its execution, it does not save and restore them for every VM exit. However, even Xen must save and restore all the VM system registers when it switches between VMs, which is a common operation on Xen because all I/O is handled in a special separate VM called Dom0. Furthermore, Xen frequently accesses the hypervisor control registers which trap when Xen is a guest hypervisor under ARMv8.3. Therefore, Xen is likely to also benefit from NEVE.

7 EVALUATION OF NEVE NESTED VIRTUALIZATION

We measured the nested virtualization performance of NEVE based on running our paravirtualized KVM/ARM guest hypervisor on our KVM/ARM host hypervisor on multicore ARM hardware. An actual hardware implementation of NEVE would not require paravirtualization and would run unmodified guest hypervisors; paravirtualization is only used to provide measurements on current ARMv8.0 hardware. We also compare NEVE against both ARMv8.3 and x86 nested virtualization. Experiments were conducted using the same hardware and software configurations as discussed in Section 5. For NEVE measurements, the guest hypervisor has been paravirtualized to use NEVE by sharing a memory region with the host hypervisor for logging the results of hypervisor instructions, and redirecting hypervisor control register accesses to the corresponding EL1 system registers, as discussed in Section 6. Although the ARM hardware we used has a GICv2 which uses a memory mapped interface

	ARMv8.3		NEVE		x86
Micro-benchmark	**Nested VM**	**Nested VM VHE**	**Nested VM**	**Nested VM VHE**	**Nested VM**
Hypercall	422,720 (155x)	307,363 (113x)	92,385 (34x)	100,895 (37x)	36,345 (31x)
Device I/O	436,924 (124x)	312,148 (88x)	96,002 (27x)	105,071 (30x)	39,108 (17x)
Virtual IPI	611,686 (73x)	494,765 (59x)	184,657 (22x)	213,256 (25x)	45,360 (16x)
Virtual EOI	71 (1x)	71 (1x)	71 (1x)	71 (1x)	316 (1x)

Table 6: Microbenchmark Cycle Counts

for registers instead of the GICv3 hypervisor control system registers discussed in Section 6, the programming interfaces for both GIC versions are almost identical.

7.1 Microbenchmark Results

We repeated the kvm-unit-test microbenchmark measurements from Section 5 using NEVE with the same nested VM configurations. Table 6 shows the results in terms of cycle counts and relative overhead compared to running in a non-nested VM, along with the previous results from Table 1. NEVE provides a dramatic performance improvement compared to ARMv8.3. When running in a nested VM, NEVE provides up to 5 times faster performance than ARMv8.3 for both non-VHE and VHE guest hypervisors. While x86 nested virtualization remains much faster in terms of absolute cycle counts, this is due to the fact that the base VM measurements are faster on x86 than on ARM. However, comparing the relative performance of a nested vs. non-nested VM on each platform, we see that a guest hypervisor using NEVE has similar overhead to x86. For example for Hypercall, NEVE incurs a 34 to 37 times slowdown while x86 incurs a 31 times slowdown running in a nested vs. non-nested VM.

Table 7 shows the average number of traps to the host hypervisor when running each microbenchmark in the nested VM. NEVE reduces the number of traps by more than six times compared to ARMv8.3. For example, Hypercall takes only one trap from a VM, but from a nested VM on ARMv8.3, it requires 126 and 82 traps to the host hypervisor using a non-VHE and VHE guest hypervisor, respectively. Using NEVE, Hypercall only requires 15 traps to the host hypervisor using either a non-VHE or VHE guest hypervisor. Although non-VHE and VHE guest hypervisors require the same number of traps for Hypercall, they incur different numbers of cycles as shown in Table 6 as the traps incurred are different with different emulation costs. For example, VHE adds an additional timer, the EL2 virtual timer, where non-VHE systems only have one virtual timer, the EL1 virtual timer. This additional timer must be supported for VHE guest hypervisors. Because of the register redirection functionality of VHE, and because

	ARMv8.3		NEVE		x86
Micro-benchmark	**Nested VM**	**Nested VM VHE**	**Nested VM**	**Nested VM VHE**	**Nested VM**
Hypercall	126	82	15	15	5
Device I/O	128	82	15	15	5
Virtual IPI	261	172	37	38	9
Virtual EOI	0	0	0	0	0

Table 7: Microbenchmark Average Trap Counts

the VHE guest hypervisor runs deprivileged in EL1, the VHE guest hypervisor directly accesses the EL1 virtual timer when it programs its EL2 virtual timer. However, when attempting to program its EL1 virtual timer, the guest hypervisor will use new VHE-specific EL02 access instructions, which always trap to the host hypervisor, resulting in traps for a VHE guest hypervisor that do not occur for a non-VHE guest hypervisor. A more optimized VHE guest hypervisor [16] with NEVE could potentially reduce the number of traps to the host hypervisor to even less than x86.

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

7.2 Application Benchmark Results

To provide a more realistic measure of performance, we next evaluated nested virtualization using widely-used CPU and I/O intensive application workloads, as listed in Table 8. We used three different configurations for our measurements: (1) native: running natively on Linux capped at 4 cores and 12 GB RAM, (2) VM: running in a 4-way SMP guest OS with 12 GB RAM using KVM as a hypervisor with 8 cores and 16 GB RAM, and (3) nested VM: running in a 4-way SMP nested guest OS with 12 GB RAM using KVM as the guest hypervisor, which is capped with 6 cores with 16 GB RAM, while the host KVM hypervisor has 8 cores and 20 GB RAM. The last two configurations are the same as those used in Section 5. For benchmarks that involve clients interacting with the server, the client ran on a separate dedicated machine and the server ran on the configuration being measured, ensuring that clients were never saturated during any of our experiments. Clients ran natively on Linux with the same kernel version and userspace as the server and configured to use the full hardware available.

Figure 2 shows the performance measurements for each VM and nested VM configuration across two different vertical scales given the large dynamic range of the measurements. Since we are most interested in overhead and in comparing

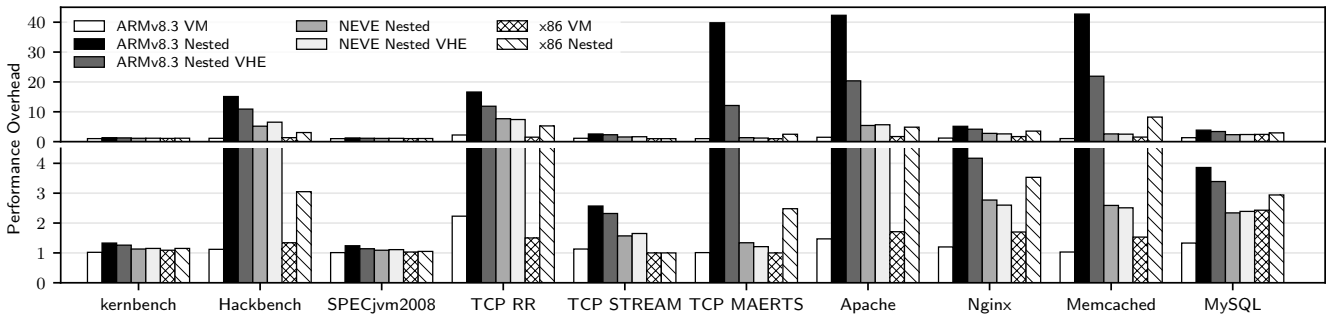

Figure 2: Application Benchmark Performance

Kernbench	Compilation of the Linux 3.17.0 kernel using the allnoconfig for ARM using GCC 4.8.2.
Hackbench	`hackbench` [40] using Unix domain sockets and 100 process groups running with 500 loops.
SPECjvm2008	`SPECjvm2008` [41] 2008 running real life applications and benchmarks chosen to measure Java Runtime Environment performance; we used 15.02 release of the Linaro AArch64 port of OpenJDK.
Netperf	`netperf` v2.6.0 [30] 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` [44] v2.3 on the remote client, measuring requests handled per second serving the 41 KB file of the GCC 4.4 manual using 10 concurrent requests.
Nginx	`Nginx` v1.4.6 Web server running `Siege` [29] v3.0.5 on the remote client, measuring requests handled per second serving the 41 KB file of the GCC 4.4 manual using 8 concurrent requests.
Memcached	`memcached` v1.4.14 using the `memtier` benchmark v1.2.3 with its default parameters.
MySQL	`MySQL` v14.14 (distrib 5.5.41) running `SysBench` v.0.4.12 using the default configuration with 200 parallel transactions.

Table 8: Application Benchmarks

across different hardware platforms, VM and nested VM performance are normalized relative to their respective ARM or x86 native execution, with lower meaning less overhead.

As expected, running in a nested VM on ARMv8.3 shows the highest overhead, in some cases more than 40 times native execution. The largest overhead occurs for network-related workloads, including Netperf TCP_MAERTS, Apache, and Memcached. The high overhead is likely due to the high frequency of interrupts caused by many incoming network packets. Injecting a high number of virtual interrupts to the nested VM results in a high number of switches between the nested VM and the guest hypervisor, which in turn results in many traps using only ARMv8.3. Hackbench also performs quite poorly as it is 15 and 11 times slower for non-VHE and VHE guest hypervisors, respectively, compared to native execution. Hackbench is a highly parallel SMP workload in which the OS frequently sends IPIs to synchronize and schedule tasks across CPU cores. As shown in Table 6, virtual IPIs are costly in nested VMs on ARMv8.3, which accounts for the noticeable slowdown in Hackbench. Compared to native execution, CPU-intensive workloads such as SPECjvm and kernbench have a relatively modest performance slowdown in nested VMs, 24% and 33% overhead for a non-VHE guest hypervisor and 14% and 26% for a VHE guest hypervisor, respectively. These workloads have much less overhead than other application workloads because they cause far fewer interactions between the nested VM and the guest hypervisor, and therefore don't suffer as much from the exit multiplication problem as network related benchmarks.

In contrast, NEVE provides significantly better ARM nested virtualization performance, reducing performance overhead by more than or close to an order of magnitude in some cases. For example, Memcached performance goes from more than a 40 times slowdown using ARMv8.3 to less than a 3 times slowdown using NEVE, more than an order of magnitude improvement. For network-related workloads including Netperf TCP MAERTS, Apache, and Memcached, NEVE successfully reduces exit multiplication by coalescing traps to reduce the performance overhead. Unlike ARMv8.3, which has significantly worse performance, NEVE provides overall performance that is comparable to, and in many cases better than, x86 nested virtualization using the latest x86 virtualization optimizations.

In fact, NEVE incurs significantly less overhead than both ARMv8.3 and x86 on many of the network-related workloads, including Netperf TCP MAERTS, Nginx, Memcached, and MySQL. MySQL runs better with NEVE because of the high cost of x86 non-nested virtualization compared to ARM, but this is not the case for the other workloads. For example, Memcached running in a nested VM on x86 shows an 8 times slowdown compared to only a 2.5 times slowdown on NEVE. The reason for this is that Memcached incurs substantially

more exits on x86 than ARM, including more than four times as many exits from the nested VM for processing I/O on x86 versus NEVE. Since the relative cost of nested VM exits is similar on x86 and NEVE as shown in Table 6, the much higher number of exits on x86 results in much higher overhead than NEVE. Netperf TCP MAERTS and Nginx exhibit similar behavior.

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

Our results are based on paravirtualizing KVM/ARM as a guest hypervisor to mimic the behavior of ARMv8.3 and NEVE on existing ARMv8.0 hardware. Future ARM hardware, such as ARMv8.3 hardware, may have somewhat different performance characteristics. In particular, ARMv8.3 is a more complex architecture than ARMv8.0, so it would not be surprising if the relative cost of traps is higher for such hardware compared to current ARMv8.0 hardware. Because NEVE improves performance by reducing the number of traps for nested virtualization, a high trap cost for actual ARMv8.3 hardware would only accentuate the performance difference between NEVE and ARMv8.3, making ARMv8.3 nested virtualization performance worse and NEVE's relative improvement even better.

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

8 RELATED WORK

Paravirtualization is used to make hypervisors simpler and faster by avoiding certain architecture features that are complex or difficult to virtualize efficiently [8, 39]. It is also used to provide virtual architectures that differ from the underlying hardware architecture and can run custom guest OSes designed for performance and scalability [46]. We leverage paravirtualization in a new way to emulate the behavior and measure the performance of new architecture features at native execution speeds on existing and currently available hardware.

Previous work has explored ways to use existing hardware to emulate new hardware. For example, Shade [13] proposed a dynamic translation framework that could run SPARCv9 binaries on a SPARCv8 CPU. However, Shade incurs significant performance overhead. Simulating SPARCv9 on SPARCv8 is more than an order of magnitude slower than native execution on SPARCv8. Our paravirtualization technique is applied statically and does not incur substantial performance overhead, but is focused on virtualization hardware support rather than emulating entire future architectures.

Much work on nested virtualization has focused on x86 [4, 10, 31, 49]. Turtles [10] was the first to show that trap-and-emulate nested virtualization provides reasonable performance on x86. Our ARM hypervisor design uses the same approach as Turtles for CPU and memory virtualization, but uses paravirtualized I/O in lieu of direct device assignment as used in Turtles; the latter was not supported on the ARM server hardware available for our measurements. However, we show that the lessons learned from trap-and-emulate nested virtualization on x86 may not apply to other architectures and that a similar approach on ARM performs poorly due to differences between the ARM and x86 virtualization support. We introduce a new architecture extension to address this problem and significantly improve ARM performance.

To optimize nested virtualization further, Intel added a new hardware extension called VMCS shadowing [27], which allows a guest hypervisor to execute VMCS access instructions without trapping. VMCS shadowing redirects instructions that are designed to access the VMCS, which is stored in in memory, to a different memory location. Our x86 measurements in Section 7.2 show that the VMCS shadowing optimization provides roughly a 10% performance improvement. Both VMCS shadowing and NEVE use the basic idea of redirection to mitigate the exit multiplication problem by reducing traps from guest hypervisors. However, unlike VMCS shadowing, NEVE introduces register redirection and rewrites system register accesses to memory accesses or to other existing registers based on a classification of the functionality of the registers. NEVE is designed for RISC

architectures without adopting techniques similar to VMCS which are more suitable for CISC architectures. Unlike the modest gain of VMCS shadowing on x86, NEVE provides an order of magnitude performance improvement on ARM. This is due in part to the CISC vs. RISC architecture designs. x86 automatically saves and restores VM state using the hardware VMCS mechanism which coalesces accesses to VM register state when changing between root and non-root mode in a single operation, mitigating the exit multiplication problem and reduces the benefit of VMCS shadowing. In contrast, ARM requires software to save and restore VM state to individual registers which results in many more accesses to VM state in software, for which NEVE can significantly reduce exit multiplication and improve performance.

Xen-Blanket [47] leverages nested virtualization to transform existing heterogeneous cloud infrastructures into a homogeneous Blanket layer to host x86 nested VMs. Unlike the aforementioned nested x86 solutions that use hardware virtualization primitives exposed to the guest hypervisor, it does not rely on the host hypervisor to expose those primitives to the nesting layers. Therefore, Xen-Blanket only supports paravirtualized guest OSes, not unmodifed OSes in the nested VM.

Agesen et al. [2] proposed software techniques for avoiding VM exits by leveraging existing work on binary translation to detect and rewrite sequences of instructions that cause multiple exits from the VM and rewrite them into translated sequences that only cause a single exit. LeVasseur et al. [34] proposed pre-virtualization, a form of static paravirtualization that uses a hypervisor-specific module in the guest OS to rewrite itself when loaded by a hypervisor. In contrast, NEVE is a hardware approach to transparently rewrite deferrable register accesses to memory accesses in the guest hypervisor and delivers substantial performance gain for workloads running in nested VMs.

Some techniques [3, 21] reduce VM exits by coalescing interrupts, effectively changing the hardware semantics to reduce interrupt overhead while increasing interrupt latency. NEVE does not defer interrupts, but defers trapping on instruction execution in a way that preserves existing architecture semantics and improves performance even in the absence of interrupts.

Various ARM virtualization approaches have been developed [5, 9, 16–18, 20, 23, 24, 26, 36, 45, 48], but none of them support nested virtualization. Our work presents the first ARM hypervisor to support nested virtualization, and introduces new architecture improvements that can be used by host hypervisors to significantly enhance performance.

As virtualization continues to be of importance, understanding the trade-offs of different approaches to hardware virtualization support is instrumental in the design of new architectures. For example, RISC-V [38] is an emerging architecture for which virtualization support is being explored. NEVE provides an important counterpoint to x86 practices and shows how acceptable nested virtualization performance can be achieved on RISC-style architectures.

9 CONCLUSIONS

We present the first in-depth study of ARM nested virtualization. We introduce a novel paravirtualization technique to evaluate the performance of new architectural features before hardware is readily available. Using this technique, we evaluate ARMv8.3 nested virtualization support and find that its performance is prohibitively expensive compared to normal virtualization, despite its similarities to x86 nested virtualization. We show how differences between ARM and x86 in non-nested virtualization support end up causing significant exit multiplication on ARM. To address this problem, we introduce NEVE, a simple architecture extension that provides register redirection, and coalesces and defers traps by logging system register accesses to memory and only copying the results of those accesses to hardware system registers when necessary. We evaluate the performance of NEVE and show that NEVE can improve nested virtualization performance by an order of magnitude on real application workloads compared to the latest ARMv8.3 architecture, and can provide up to three times less virtualization overhead than x86. NEVE is straightforward to implement in ARM and will be included the next version of the ARM architecture.

10 ACKNOWLEDGMENTS

Martha Kim provided helpful comments on earlier drafts of this paper. This work was supported in part by ARM and NSF grants CNS-1717801, CNS-1563555, CNS-1422909, and CCF-1162021.

REFERENCES

[1] Keith Adams and Ole Agesen. 2006. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, New York, NY, USA, 2–13. https://doi.org/10.1145/1168857.1168860

[2] Ole Agesen, Jim Mattson, Radu Rugina, and Jeffrey Sheldon. 2012. Software Techniques for Avoiding Hardware Virtualization Exits. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC '12)*. USENIX Association, Berkeley, CA, USA, 35–35. http://dl.acm.org/citation.cfm?id=2342821.2342856

[3] Irfan Ahmad, Ajay Gulati, and Ali Mashtizadeh. 2011. vIC: Interrupt Coalescing for Virtual Machine Storage Device IO. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC '11)*. USENIX Association, Berkeley, CA, USA, 4–4. http://dl.acm.org/citation.cfm?id=2002181.2002185

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

[5] Jeremy Andrus, Christoffer Dall, Alexander Van't Hof, Oren Laadan, and Jason Nieh. 2011. Cells: A Virtual Mobile Smartphone Architecture. http://doi.acm.org/10.1145/2043556.2043574. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*. 173–187.

[6] ARM Ltd. 2011. ARM Generic Interrupt Controller Architecture version 2.0 ARM IHI 0048B. (June 2011).

[7] ARM Ltd. 2013. ARM Architecture Reference Manual ARMv8-A DDI0487A.a. (Sept. 2013).

[8] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. ACM, New York, NY, USA, 164–177. https://doi.org/10.1145/945445.945462

[9] Ken Barr, Prashanth Bungale, Stephen Deasy, Viktor Gyuris, Perry Hung, Craig Newell, Harvey Tuch, and Bruno Zoppis. 2010. The VMware Mobile Virtualization Platform: Is That a Hypervisor in Your Pocket? *SIGOPS Oper. Syst. Rev.* 44, 4 (Dec. 2010), 124–135. https://doi.org/10.1145/1899928.1899945

[10] 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 USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 1–6. http://dl.acm.org/citation.cfm?id=1924943.1924973

[11] David Brash. 2016. ARMv8-A Architecture - 2016 Additions. (Oct. 2016). https://community.arm.com/groups/processors/blog/2016/10/27/armv8-a-architecture-2016-additions.

[12] CloudShare. 2017. Is your virtualized infrastructure keeping you from the cloud? (2017). https://www.cloudshare.com/technology/nested-virtualization/.

[13] Bob Cmelik and David Keppel. 1994. Shade: A Fast Instruction-set Simulator for Execution Profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '94)*. ACM, New York, NY, USA, 128–137. https://doi.org/10.1145/183018.183032

[14] Ravello Community. 2016. Nested virtualization: How to run nested KVM on AWS or Google Cloud. (Jan. 2016). https://blogs.oracle.com/ravello/run-nested-kvm-on-aws-google.

[15] 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 '16)*. IEEE Press, Piscataway, NJ, USA, 304–316. https://doi.org/10.1109/ISCA.2016.35

[16] 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 Conference on Annual Technical Conference (USENIX ATC '17)*. USENIX Association, Berkeley, CA, USA, 221–233.

[17] Christoffer Dall and Jason Nieh. 2010. KVM for ARM. In *Proceedings of the Ottawa Linux Symposium*. 45–56.

[18] 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 '14)*. ACM, New York, NY, USA, 333–348. https://doi.org/10.1145/2541940.2541946

[19] Dina Bass and Ian King. 2017. Microsoft Pledges to Use ARM Server Chips, Threatening Intel's Dominance. (March 2017). https://www.bloomberg.com/news/articles/2017-03-08/microsoft-pledges-to-use-arm-server-chips-threatening-intel-s-dominance.

[20] Jiun-Hung Ding, Chang-Jung Lin, Ping-Hao Chang, Chieh-Hao Tsang, Wei-Chung Hsu, and Yeh-Ching Chung. 2012. ARMvisor: System Virtualization for ARM. In *Proceedings of the Ottawa Linux Symposium*. 93–107.

[21] Yaozu Dong, Dongxiao Xu, Yang Zhang, and Guangdeng Liao. 2011. Optimizing network I/O virtualization with efficient interrupt coalescing and virtual receive side scaling. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*. IEEE, 26–34.

[22] Joy Fan. 2017. Nested Virtualization in Azure. (July 2017). https://azure.microsoft.com/en-us/blog/nested-virtualization-in-azure/.

[23] General Dynamics. 2013. OKL4 Microvisor. (Feb. 2013). http://www.ok-labs.com/products/okl4-microvisor.

[24] Green Hills Software. 2014. INTEGRITY Secure Virtualization. (Jan. 2014). http://www.ghs.com/products/rtos/integrity_virtualization.html.

[25] Stefan Hajnoczi. 2011. An Updated Overview of the QEMU Storage Stack. (June 2011). https://events.linuxfoundation.org/slides/2011/linuxcon-japan/lcj2011_hajnoczi.pdf.

[26] J.Y Hwang, S.B Suh, S.K Heo, C.J Park, J.M Ryu, S.Y Park, and C.R Kim. 2008. Xen on ARM: System Virtualization using Xen Hypervisor for ARM-based Secure Mobile Phones. In *Proceedings of the 5th Consumer Communications and Newtork Conference*. 257–261.

[27] Intel. 2013. 4th Generation Intel Core vPro Processors with Intel VMCS Shadowing. (2013). http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-vmcs-shadowing-paper.pdf.

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

[29] Jeffrey Fulmer. 2012. Siege. (Jan. 2012). https://www.joedog.org/siege-home/.

[30] Rick Jones. 2016. Netperf. (Nov. 2016). http://www.netperf.org/netperf/.

[31] Bernhard Kauer, Paulo Verissimo, and Alysson Bessani. 2011. Recursive Virtual Machines for Advanced Security Mechanisms. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSNW '11)*. IEEE Computer Society, Washington, DC, USA, 117–122. https://doi.org/10.1109/DSNW.2011.5958796

[32] KVM contributors. 2015. Tuning KVM. (May 2015). http://www.linux-kvm.org/page/Tuning_KVM.

[33] KVM contributors. 2017. KVM Unit Tests. (May 2017). http://www.linux-kvm.org/page/KVM-unit-tests.

[34] Joshua LeVasseur, Volkmar Uhlig, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. 2005. *Pre-Virtualization: Slashing the Cost of Virtualization.* Technical Report 2005-30. Fakultät für Informatik, Universität Karlsruhe (TH).

[35] Gerald J. Popek and Robert P. Goldberg. 1974. Formal Requirements for Virtualizable Third Generation Architectures. *Commun. ACM* 17, 7 (July 1974), 412–421. https://doi.org/10.1145/361011.361073

[36] Red Bend Software. 2013. vLogix Mobile. (Feb. 2013). http://www.redbend.com/en/mobile-virtualization.

[37] 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). https://www.usenix.org/publications/login/dec14/ricci

[38] RISC-V Foundation. 2017. RISC-V. (2017). http://www.riscv.org.

[39] Rusty Russell. 2008. Virtio: Towards a De-facto Standard for Virtual I/O Devices. *SIGOPS Oper. Syst. Rev.* 42, 5 (July 2008), 95–103. https://doi.org/10.1145/1400097.1400108

[40] Rusty Russell, Yanmin Zhang, Ingo Molnar, and David Sommerseth. 2008. Hackbench. (Jan. 2008). http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c.

[41] Standard Performance Evaluation Corporation. 2015. SPECjvm2008. (Nov. 2015). https://www.spec.org/jvm2008.

[42] SUSE. 2016. Performance Implications of Cache Modes. (Sept. 2016). https://www.suse.com/documentation/sles11/book_kvm/data/sect1_3_chapter_book_kvm.html.

[43] Ravello Systems. 2017. Run VMware workloads on public clouds - without any changes. (2017). https://www.ravellosystems.com/.

[44] The Apache Software Foundation. 2015. ab - Apache HTTP server benchmarking tool. (April 2015). http://httpd.apache.org/docs/2.4/programs/ab.html.

[45] Prashant Varanasi and Gernot Heiser. 2011. Hardware-supported Virtualization on ARM. In *Proceedings of the Second Asia-Pacific Workshop on Systems (APSys '11)*. ACM, New York, NY, USA, Article 11, 5 pages. https://doi.org/10.1145/2103799.2103813

[46] 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 '02)*. USENIX Association, Berkeley, CA, USA, 195–209. http://dl.acm.org/citation.cfm?id=1060289.1060308

[47] Dan Williams, Hani Jamjoom, and Hakim Weatherspoon. 2012. The Xen-Blanket: Virtualize Once, Run Everywhere. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. ACM, New York, NY, USA, 113–126. https://doi.org/10.1145/2168836.2168849

[48] Xen ARM with Virtualization Extensions. 2015. http://wiki.xenproject.org/wiki/Xen_ARM_with_Virtualization_Extensions. (April 2015).

[49] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. 2011. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 203–216. https://doi.org/10.1145/2043556.2043576