2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture

ARM Virtualization: Performance and Architectural Implications

Christoffer Dall, Shih-Wei Li, Jin Tack Lim, Jason Nieh, and Georgios Koloventzos
Department of Computer Science
Columbia University
New York, NY, United States
{cdall,shihwei,jintack,nieh,gkoloven}@cs.columbia.edu

***Abstract*—ARM servers are becoming increasingly common, making server technologies such as virtualization for ARM of growing importance. We present the first study of ARM virtualization performance on server hardware, including multi-core measurements of two popular ARM and x86 hypervisors, KVM and Xen. We show how ARM hardware support for virtualization can enable much faster transitions between VMs and the hypervisor, a key hypervisor operation. However, current hypervisor designs, including both Type 1 hypervisors such as Xen and Type 2 hypervisors such as KVM, are not able to leverage this performance benefit for real application workloads. We discuss the reasons why and show that other factors related to hypervisor software design and implementation have a larger role in overall performance. Based on our measurements, we discuss changes to ARM's hardware virtualization support that can potentially bridge the gap to bring its faster VM-to-hypervisor transition mechanism to modern Type 2 hypervisors running real applications. These changes have been incorporated into the latest ARM architecture.**

***Keywords*-computer architecture; hypervisors; operating systems; virtualization; multi-core; performance; ARM; x86**

I. Introduction

ARM CPUs have become the platform of choice across mobile and embedded systems, leveraging their benefits in customizability and power efficiency in these markets. The release of the 64-bit ARM architecture, ARMv8 [1], with its improved computing capabilities is spurring an upward push of ARM CPUs into traditional server systems. A growing number of companies are deploying commercially available ARM servers to meet their computing infrastructure needs. As virtualization plays an important role for servers, ARMv8 provides hardware virtualization support. Major virtualization players, including KVM [2] and Xen [3], leverage ARM hardware virtualization extensions to support unmodified existing operating systems (OSes) and applications with improved hypervisor performance.

Despite these trends and the importance of ARM virtualization, little is known in practice regarding how well virtualized systems perform using ARM. There are no detailed studies of ARM virtualization performance on server hardware. Although KVM and Xen both have ARM and x86 virtualization solutions, there are substantial differences between their ARM and x86 approaches because of key architectural differences between the underlying ARM and x86 hardware virtualization mechanisms. It is unclear whether these differences have a material impact, positive or negative, on performance. The lack of clear performance data limits the ability of hardware and software architects to build efficient ARM virtualization solutions, and limits the ability of companies to evaluate how best to deploy ARM virtualization solutions to meet their infrastructure needs. The increasing demand for ARM-based solutions and growing investments in ARM server infrastructure makes this problem one of key importance.

We present the first in-depth study of ARM virtualization performance on multi-core server hardware. We measure the performance of the two most popular ARM hypervisors, KVM and Xen, and compare them with their respective x86 counterparts. These hypervisors are important and useful to compare on ARM given their popularity and their different design choices. Xen is a standalone bare-metal hypervisor, commonly referred to as a Type 1 hypervisor. KVM is a hosted hypervisor integrated within an existing OS kernel, commonly referred to as a Type 2 hypervisor.

We have designed and run a number of microbenchmarks to analyze the performance of frequent low-level hypervisor operations, and we use these results to highlight differences in performance between Type 1 and Type 2 hypervisors on ARM. A key characteristic of hypervisor performance is the cost of transitioning from a virtual machine (VM) to the hypervisor, for example to process interrupts, allocate memory to the VM, or perform I/O. We show that Type 1 hypervisors, such as Xen, can transition between the VM and the hypervisor much faster than Type 2 hypervisors, such as KVM, on ARM. We show that ARM can enable significantly faster transitions between the VM and a Type 1 hypervisor compared to x86. On the other hand, Type 2 hypervisors such as KVM, incur much higher overhead on ARM for VM-to-hypervisor transitions compared to x86. We also show that for some more complicated hypervisor operations, such as switching between VMs, Type 1 and Type 2 hypervisors perform equally fast on ARM.

Despite the performance benefit in VM transitions that ARM can provide, we show that current hypervisor designs, including both KVM and Xen on ARM, result in real application performance that cannot be easily correlated with the low-level virtualization operation performance. In fact, for many workloads, we show that KVM ARM, a

1063-6897/16 $31.00 © 2016 IEEE
DOI 10.1109/ISCA.2016.35

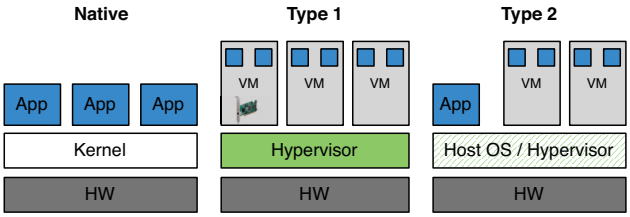

Figure 1: Hypervisor Design

Type 2 hypervisor, can meet or exceed the performance of Xen ARM, a Type 1 hypervisor, despite the faster transitions between the VM and hypervisor using Type 1 hypervisor designs on ARM. We show how other factors related to hypervisor software design and implementation play a larger role in overall performance. These factors include the hypervisor's virtual I/O model, the ability to perform zero copy I/O efficiently, and interrupt processing overhead. Although ARM hardware virtualization support incurs higher overhead on VM-to-hypervisor transitions for Type 2 hypervisors than x86, we show that both types of ARM hypervisors can achieve similar, and in some cases lower, performance overhead than their x86 counterparts on real application workloads.

To enable modern hypervisor designs to leverage the potentially faster VM transition costs when using ARM hardware, we discuss changes to the ARMv8 architecture that can benefit Type 2 hypervisors. These improvements potentially enable Type 2 hypervisor designs such as KVM to achieve faster VM-to-hypervisor transitions, including for hypervisor events involving I/O, resulting in reduced virtualization overhead on real application workloads. ARM has incorporated these changes into the latest ARMv8.1 architecture.

II. Background

Hypervisor Overview. Figure 1 depicts the two main hypervisor designs, Type 1 and Type 2. Type 1 hypervisors, like Xen, comprise a separate hypervisor software component, which runs directly on the hardware and provides a virtual machine abstraction to VMs running on top of the hypervisor. Type 2 hypervisors, like KVM, run an existing OS on the hardware and run both VMs and applications on top of the OS. Type 2 hypervisors typically modify the existing OS to facilitate running of VMs, either by integrating the Virtual Machine Monitor (VMM) into the existing OS source code base, or by installing the VMM as a driver into the OS. KVM integrates directly with Linux [4] where other solutions such as VMware Workstation [5] use a loadable driver in the existing OS kernel to monitor virtual machines. The OS integrated with a Type 2 hypervisor is commonly referred to as the host OS, as opposed to the guest OS which runs in a VM.

One advantage of Type 2 hypervisors over Type 1 hypervisors is the reuse of existing OS code, specifically device drivers for a wide range of available hardware. This is especially true for server systems with PCI where any commercially available PCI adapter can be used. Traditionally, a Type 1 hypervisor suffers from having to re-implement device drivers for all supported hardware. However, Xen [6], a Type 1 hypervisor, avoids this by only implementing a minimal amount of hardware support directly in the hypervisor and running a special privileged VM, Dom0, which runs an existing OS such as Linux and uses all the existing device drivers for that OS. Xen then uses Dom0 to perform I/O using existing device drivers on behalf of normal VMs, also known as DomUs.

Transitions from a VM to the hypervisor occur whenever the hypervisor exercises system control, such as processing interrupts or I/O. The hypervisor transitions back to the VM once it has completed its work managing the hardware, letting workloads in VMs continue executing. The cost of such transitions is pure overhead and can add significant latency in communication between the hypervisor and the VM. A primary goal in designing both hypervisor software and hardware support for virtualization is to reduce the frequency and cost of transitions as much as possible.

VMs can run guest OSes with standard device drivers for I/O, but because they do not have direct access to hardware, the hypervisor would need to emulate real I/O devices in software. This results in frequent transitions between the VM and the hypervisor, making each interaction with the emulated device an order of magnitude slower than communicating with real hardware. Alternatively, direct passthrough of I/O from a VM to the real I/O devices can be done using device assignment, but this requires more expensive hardware support and complicates VM migration.

Instead, the most common approach is paravirtual I/O in which custom device drivers are used in VMs for virtual devices supported by the hypervisor. The interface between the VM device driver and the virtual device is specifically designed to optimize interactions between the VM and the hypervisor and facilitate fast I/O. KVM uses an implementation of the Virtio [7] protocol for disk and networking support, and Xen uses its own implementation referred to simply as *Xen PV*. In KVM, the virtual device backend is implemented in the host OS, and in Xen the virtual device backend is implemented in the Dom0 kernel. A key potential performance advantage for KVM is that the virtual device implementation in the KVM host kernel has full access to all of the machine's hardware resources, including VM memory. On the other hand, Xen provides stronger isolation between the virtual device implementation and the VM as the Xen virtual device implementation lives in a separate VM, Dom0, which only has access to memory and hardware resources specifically allocated to it by the Xen hypervisor.

ARM Virtualization Extensions. To enable hypervisors to efficiently run VMs with unmodified guest OSes, ARM introduced hardware virtualization extensions [1] to over-

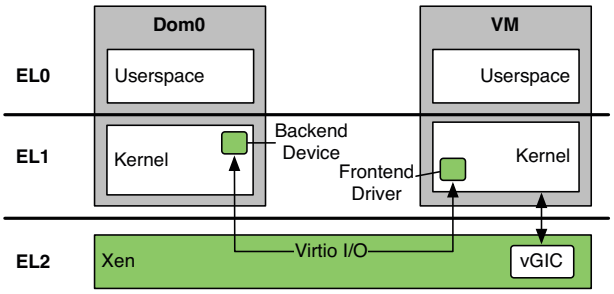

Figure 2: Xen ARM Architecture

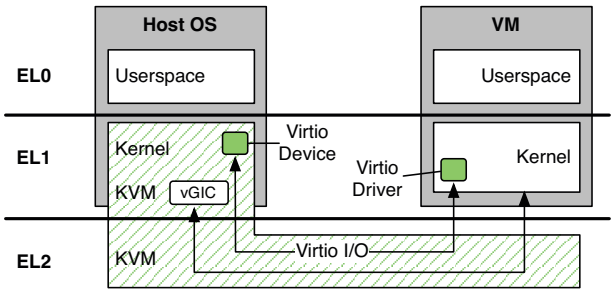

Figure 3: KVM ARM Architecture

come the limitation that the ARM architecture was not classically virtualizable [8]. All server and networking class ARM hardware is expected to implement these extensions. We provide a brief overview of the ARM hardware virtualization extensions and how hypervisors leverage these extensions, focusing on ARM CPU virtualization support and contrasting it to how x86 works.

The ARM virtualization extensions are centered around a new CPU privilege level (also known as *exception level*), EL2, added to the existing user and kernel levels, EL0 and EL1, respectively. Software running in EL2 can configure the hardware to support VMs. To allow VMs to interact with an interface identical to the physical machine while isolating them from the rest of the system and preventing them from gaining full access to the hardware, a hypervisor enables the virtualization features in EL2 before switching to a VM. The VM will then execute normally in EL0 and EL1 until some condition is reached that requires intervention of the hypervisor. At this point, the hardware traps into EL2 giving control to the hypervisor, which can then interact directly with the hardware and eventually return to the VM again. When all virtualization features are disabled in EL2, software running in EL1 and EL0 works just like on a system without the virtualization extensions where software running in EL1 has full access to the hardware.

ARM hardware virtualization support enables traps to EL2 on certain operations, enables virtualized physical memory support, and provides virtual interrupt and timer support. ARM provides CPU virtualization by allowing software in EL2 to configure the CPU to trap to EL2 on sensitive instructions that cannot be safely executed by a VM. ARM provides memory virtualization 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. When Stage-2 translation is enabled, the ARM architecture defines three address spaces: Virtual Addresses (VA), Intermediate Physical Addresses (IPA), and Physical Addresses (PA). Stage-2 translation, configured in EL2, translates from IPAs to PAs. ARM provides interrupt virtualization through a set of virtualization extensions to the ARM Generic Interrupt Controller (GIC) architecture, which allows a hypervisor to program the GIC to inject virtual interrupts to VMs, which VMs can acknowledge and complete without trapping to the hypervisor. However, enabling and disabling virtual interrupts must be done in EL2. Furthermore, all physical interrupts are taken to EL2 when running in a VM, and therefore must be handled by the hypervisor. Finally, ARM provides a virtual timer, which can be configured by the VM without trapping to the hypervisor. However, when the virtual timer fires, it raises a physical interrupt, which must be handled by the hypervisor and translated into a virtual interrupt.

ARM hardware virtualization support has some similarities to x86[1], including providing a means to trap on sensitive instructions and a nested set of page tables to virtualize physical memory. However, there are key differences in how they support Type 1 and Type 2 hypervisors. While ARM virtualization extensions are centered around a separate CPU mode, x86 support provides a mode switch, root vs. non-root mode, completely orthogonal from the CPU privilege rings. While ARM's EL2 is a strictly different CPU mode with its own set of features, x86 root mode supports the same full range of user and kernel mode functionality as its non-root mode. Both ARM and x86 trap into their respective EL2 and root modes, but transitions between root and non-root mode on x86 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, being a RISC-style architecture, instead has a simpler hardware mechanism to transition between EL1 and EL2 but leaves it up to software to decide which state needs to be saved and restored. This provides more flexibility in the amount of work that needs to be done when transitioning between EL1 and EL2 compared to switching between root and non-root mode on x86, but poses different requirements on hypervisor software implementation.

ARM Hypervisor Implementations. As shown in Figures 2 and 3, Xen and KVM take different approaches to using ARM hardware virtualization support. Xen as a Type 1 hypervisor design maps easily to the ARM architecture, run-

[1]Since Intel's and AMD's hardware virtualization support are very similar, we limit our comparison to ARM and Intel.

ning the entire hypervisor in EL2 and running VM userspace and VM kernel in EL0 and EL1, respectively. However, existing OSes are designed to run in EL1, so a Type 2 hypervisor that leverages an existing OS such as Linux to interface with hardware does not map as easily to the ARM architecture. EL2 is strictly more privileged and a separate CPU mode with different registers than EL1, so running Linux in EL2 would require substantial changes to Linux that would not be acceptable in practice. KVM instead runs across both EL2 and EL1 using split-mode virtualization [2], sharing EL1 between the host OS and VMs and running a minimal set of hypervisor functionality in EL2 to be able to leverage the ARM virtualization extensions. KVM enables virtualization features in EL2 when switching from the host to a VM, and disables them when switching back, allowing the host full access to the hardware from EL1 and properly isolating VMs also running in EL1. As a result, transitioning between the VM and the hypervisor involves transitioning to EL2 to run the part of KVM running in EL2, then transitioning to EL1 to run the rest of KVM and the host kernel. However, because both the host and the VM run in EL1, the hypervisor must context switch all register state when switching between host and VM execution context, similar to a regular process context switch.

This difference on ARM between Xen and KVM does not exist on x86 because the root mode used by the hypervisor does not limit or change how CPU privilege levels are used. Running Linux in root mode does not require any changes to Linux, so KVM maps just as easily to the x86 architecture as Xen by running the hypervisor in root mode.

KVM only runs the minimal set of hypervisor functionality in EL2 to be able to switch between VMs and the host and emulates all virtual device in the host OS running in EL1 and EL0. When a KVM VM performs I/O it involves trapping to EL2, switching to host EL1, and handling the I/O request in the host. Because Xen only emulates the GIC in EL2 and offloads all other I/O handling to Dom0, when a Xen VM performs I/O, it involves trapping to the hypervisor, signaling Dom0, scheduling Dom0, and handling the I/O request in Dom0.

III. Experimental Design

To evaluate the performance of ARM virtualization, we ran both microbenchmarks and real application workloads on the most popular hypervisors on ARM server hardware. As a baseline for comparison, we also conducted the same experiments with corresponding x86 hypervisors and server hardware. We leveraged the CloudLab [9] infrastructure for both ARM and x86 hardware.

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 Dell PowerEdge r320 servers, each with a 64-bit Xeon 2.1 GHz ES-2450 with 8 physical CPU cores. Hyperthreading was disabled on the r320 nodes to provide a similar hardware configuration to the ARM servers. Each r320 node had 16 GB of RAM, a 4x500 GB 7200 RPM SATA RAID5 HD for storage, and a Dual-port Mellanox MX354A 10 GbE NIC. All servers are connected via 10 GbE, and the interconnecting network switch [10] easily handles multiple sets of nodes communicating with full 10 Gb bandwidth such that experiments involving networking between two nodes can be considered isolated and unaffected by other traffic in the system. Using 10 Gb Ethernet was important, as many benchmarks were unaffected by virtualization when run over 1 Gb Ethernet, because the network itself became the bottleneck.

To provide comparable measurements, we kept the software environments across all hardware platforms and all hypervisors the same as much as possible. We used the most recent stable versions available at the time of our experiments of the most popular hypervisors on ARM and their counterparts on x86: KVM in Linux 4.0-rc4 with QEMU 2.2.0, and Xen 4.5.0. KVM was configured with its standard VHOST networking feature, allowing data handling to occur in the kernel instead of userspace, and with `cache=none` for its block storage devices. Xen was configured with its in-kernel block and network backend drivers to provide best performance and reflect the most commonly used I/O configuration for Xen deployments. Xen x86 was configured to use HVM domains, except for Dom0 which was only supported as a PV instance. All hosts and VMs used Ubuntu 14.04 with the same Linux 4.0-rc4 kernel and software configuration for all machines. A few patches were applied to support the various hardware configurations, such as adding support for the APM X-Gene PCI bus for the HP m400 servers. All VMs used paravirtualized I/O, typical of cloud infrastructure deployments such as Amazon EC2, instead of device passthrough, due to the absence of an IOMMU in our test environment.

We ran benchmarks both natively on the hosts and in VMs. Each physical or virtual machine instance used for running benchmarks was configured as a 4-way SMP with 12 GB of RAM to provide a common basis for comparison. This involved three configurations: (1) running natively on Linux capped at 4 cores and 12 GB RAM, (2) running in a VM using KVM with 8 cores and 16 GB RAM with the VM capped at 4 virtual CPUs (VCPUs) and 12 GB RAM, and (3) running in a VM using Xen with Dom0, the privileged domain used by Xen with direct hardware access, capped at 4 VCPUs and 4 GB RAM and the VM capped at 4 VCPUs and 12 GB RAM. Because KVM configures the total hardware available while Xen configures the hardware dedicated to Dom0, the configuration parameters are different but the effect is the same, which is to leave the hypervisor with 4 cores and 4 GB RAM to use outside of what is used

by the VM. We use and measure multi-core configurations to reflect real-world server deployments. The memory limit was used to ensure a fair comparison across all hardware configurations given the RAM available on the x86 servers and the need to also provide RAM for use by the hypervisor when running VMs. For benchmarks that involve clients interfacing with the server, the clients were run natively on Linux and configured to use the full hardware available.

To improve precision of our measurements and for our experimental setup to mimic recommended configuration best practices [11], we pinned each VCPU to a specific physical CPU (PCPU) and generally ensured that no other work was scheduled on that PCPU. In KVM, all of the host's device interrupts and processes were assigned to run on a specific set of PCPUs and each VCPU was pinned to a dedicated PCPU from a separate set of PCPUs. In Xen, we configured Dom0 to run on a set of PCPUs and DomU to run a separate set of PCPUs. We further pinned each VCPU of both Dom0 and DomU to its own PCPU.

IV. Microbenchmark Results

We designed and ran a number of microbenchmarks to quantify important low-level interactions between the hypervisor and the ARM hardware support for virtualization. A primary performance cost in running in a VM is how much time must be spent outside the VM, which is time not spent running the workload in the VM and therefore is virtualization overhead compared to native execution. Therefore, our microbenchmarks are designed to measure time spent handling a trap from the VM to the hypervisor, including time spent on transitioning between the VM and the hypervisor, time spent processing interrupts, time spent switching between VMs, and latency added to I/O.

We designed a custom Linux kernel driver, which ran in the VM under KVM and Xen, on ARM and x86, and executed the microbenchmarks in the same way across all platforms. Measurements were obtained using cycle counters and ARM hardware timer counters to ensure consistency across multiple CPUs. Instruction barriers were used before and after taking timestamps to avoid out-of-order execution or pipelining from skewing our measurements.

Because these measurements were at the level of a few hundred to a few thousand cycles, it was important to minimize measurement variability, especially in the context of measuring performance on multi-core systems. Variations caused by interrupts and scheduling can skew measurements by thousands of cycles. To address this, we pinned and isolated VCPUs as described in Section III, and also ran these measurements from within VMs pinned to specific VCPUs, assigning all virtual interrupts to other VCPUs.

Using this framework, we ran seven microbenchmarks that measure various low-level aspects of hypervisor performance, as listed in Table I. Table II presents the results from running these microbenchmarks on both ARM and x86 server hardware. Measurements are shown in cycles instead of time to provide a useful comparison across server hardware with different CPU frequencies, but we focus our analysis on the ARM measurements.

Name	Description
Hypercall	Transition from VM to hypervisor and return to VM without doing any work in the hypervisor. Measures bidirectional base transition cost of hypervisor operations.
Interrupt Controller Trap	Trap from VM to emulated interrupt controller then return to VM. Measures a frequent operation for many device drivers and baseline for accessing I/O devices emulated in the hypervisor.
Virtual IPI	Issue a virtual IPI from a VCPU to another VCPU running on a different PCPU, both PCPUs executing VM code. Measures time between sending the virtual IPI until the receiving VCPU handles it, a frequent operation in multi-core OSes.
Virtual IRQ Completion	VM acknowledging and completing a virtual interrupt. Measures a frequent operation that happens for every injected virtual interrupt.
VM Switch	Switch from one VM to another on the same physical core. Measures a central cost when oversubscribing physical CPUs.
I/O Latency Out	Measures latency between a driver in the VM signaling the virtual I/O device in the hypervisor and the virtual I/O device receiving the signal. For KVM, this traps to the host kernel. For Xen, this traps to Xen then raises a virtual interrupt to Dom0.
I/O Latency In	Measures latency between the virtual I/O device in the hypervisor signaling the VM and the VM receiving the corresponding virtual interrupt. For KVM, this signals the VCPU thread and injects a virtual interrupt for the Virtio device. For Xen, this traps to Xen then raises a virtual interrupt to DomU.

Table I: Microbenchmarks

The Hypercall microbenchmark shows that transitioning from a VM to the hypervisor on ARM can be significantly faster than x86, as shown by the Xen ARM measurement, which takes less than a third of the cycles that Xen or KVM on x86 take. As explained in Section II, the ARM architecture provides a separate CPU mode with its own register bank to run an isolated Type 1 hypervisor like Xen. Transitioning from a VM to a Type 1 hypervisor requires little more than context switching the general purpose registers as running the two separate execution contexts, VM and the hypervisor, is supported by the separate ARM hardware state for EL2. While ARM implements additional register state to support the different execution context of the hypervisor, x86 transitions from a VM to the hypervisor by switching from non-root to root mode which requires context switching the entire CPU register state to the VMCS in memory, which is much more expensive even with hardware support.

However, the Hypercall microbenchmark also shows that transitioning from a VM to the hypervisor is more than an order of magnitude more expensive for Type 2 hypervisors like KVM than for Type 1 hypervisors like Xen. This is

	ARM		x86	
Microbenchmark	**KVM**	**Xen**	**KVM**	**Xen**
Hypercall	6,500	376	1,300	1,228
Interrupt Controller Trap	7,370	1,356	2,384	1,734
Virtual IPI	11,557	5,978	5,230	5,562
Virtual IRQ Completion	71	71	1,556	1,464
VM Switch	10,387	8,799	4,812	10,534
I/O Latency Out	6,024	16,491	560	11,262
I/O Latency In	13,872	15,650	18,923	10,050

Table II: Microbenchmark Measurements (cycle counts)

Register State	**Save**	**Restore**
GP Regs	152	184
FP Regs	282	310
EL1 System Regs	230	511
VGIC Regs	3,250	181
Timer Regs	104	106
EL2 Config Regs	92	107
EL2 Virtual Memory Regs	92	107

Table III: KVM ARM Hypercall Analysis (cycle counts)

because although all VM traps are handled in EL2, a Type 2 hypervisor is integrated with a host kernel and both run in EL1. This results in four additional sources of overhead. First, transitioning from the VM to the hypervisor involves not only trapping to EL2, but also returning to the host OS in EL1, as shown in Figure 3, incurring a double trap cost. Second, because the host OS and the VM both run in EL1 and ARM hardware does not provide any features to distinguish between the host OS running in EL1 and the VM running in EL1, software running in EL2 must context switch all the EL1 system register state between the VM guest OS and the Type 2 hypervisor host OS, incurring added cost of saving and restoring EL1 register state. Third, because the host OS runs in EL1 and needs full access to the hardware, the hypervisor must disable traps to EL2 and Stage-2 translation from EL2 while switching from the VM to the hypervisor, and enable them when switching back to the VM again. Fourth, because the Type 2 hypervisor runs in EL1 but needs to access VM control register state such as the VGIC state, which can only be accessed from EL2, there is additional overhead to read and write the VM control register state in EL2. There are two approaches. One, the hypervisor can jump back and forth between EL1 and EL2 to access the control register state when needed. Two, it can copy the full register state to memory while it is still in EL2, return to the host OS in EL1 and read and write the memory copy of the VM control state, and then finally copy the state from memory back to the EL2 control registers when the hypervisor is running in EL2 again. Both methods incur much overhead, but the first makes the software implementation complicated and difficult to maintain. KVM ARM currently takes the second approach of reading and writing all VM control registers in EL2 during each transition between the VM and the hypervisor.

While the cost of the trap between CPU modes itself is not very high as shown in previous work [2], our measurements show that there is a substantial cost associated with saving and restoring register state to switch between EL2 and the host in EL1. Table III provides a breakdown of the cost of context switching the relevant register state when performing the Hypercall microbenchmark measurement on KVM ARM. Context switching consists of saving register state to memory and restoring the new context's state from memory to registers. The cost of saving and restoring this state accounts for almost all of the Hypercall time, indicating that context switching state is the primary cost due to KVM ARM's design, not the cost of extra traps. Unlike Xen ARM which only incurs the relatively small cost of saving and restoring the general-purpose (GP) registers, KVM ARM saves and restores much more register state at much higher cost. Note that for ARM, the overall cost of saving register state, when transitioning from a VM to the hypervisor, is much more expensive than restoring it, when returning back to the VM from the hypervisor, due to the cost of reading the VGIC register state.

Unlike on ARM, both x86 hypervisors spend a similar amount of time transitioning from the VM to the hypervisor. Since both KVM and Xen leverage the same x86 hardware mechanism for transitioning between the VM and the hypervisor, they have similar performance. Both x86 hypervisors run in root mode and run their VMs in non-root mode, and switching between the two modes involves switching a substantial portion of the CPU register state to the VMCS in memory. Switching this state to memory is fast on x86, because it is performed by hardware in the context of a trap or as a result of executing a single instruction. In contrast, ARM provides a separate CPU mode for the hypervisor with separate registers, and ARM only needs to switch state to memory when running a different execution context in EL1. ARM can be much faster, as in the case of Xen ARM which does its hypervisor work in EL2 and does not need to context switch much register state, or it can be much slower, as in the case of KVM ARM which context switches more register state without the benefit of hardware support like x86.

The large difference in the cost of transitioning between the VM and hypervisor between Type 1 and Type 2 hypervisors results in Xen ARM being significantly faster at handling interrupt related traps, because Xen ARM emulates the ARM GIC interrupt controller directly in the hypervisor running in EL2 as shown in Figure 2. In contrast, KVM ARM emulates the GIC in the part of the hypervisor running in EL1. Therefore, operations such as accessing registers in the emulated GIC, sending virtual IPIs, and receiving virtual interrupts are much faster on Xen ARM than KVM ARM. This is shown in Table II in the measurements for the Interrupt Controller trap and Virtual IPI microbenchmarks, in which Xen ARM is faster than KVM ARM by roughly the same difference as for the Hypercall microbenchmark.

However, Table II shows that for the remaining mi-

crobenchmarks, Xen ARM does not enjoy a large performance advantage over KVM ARM and in fact does worse for some of the microbenchmarks. The reasons for this differ from one microbenchmark to another: For the Virtual IRQ Completion microbenchmark, both KVM ARM and Xen ARM are very fast because the ARM hardware includes support for completing interrupts directly in the VM without trapping to the hypervisor. The microbenchmark runs much faster on ARM than x86 because the latter has to trap to the hypervisor. More recently, vAPIC support has been added to x86 with similar functionality to avoid the need to trap to the hypervisor so that newer x86 hardware with vAPIC support should perform more comparably to ARM [12].

For the VM Switch microbenchmark, Xen ARM is only slightly faster than KVM ARM because both hypervisor implementations have to context switch the state between the VM being switched out and the one being switched in. Unlike the Hypercall microbenchmark where only KVM ARM needed to context switch EL1 state and per VM EL2 state, in this case both KVM and Xen ARM need to do this, and Xen ARM therefore does not directly benefit from its faster VM-to-hypervisor transition. Xen ARM is still slightly faster than KVM, however, because to switch between VMs, Xen ARM simply traps to EL2 and performs a single context switch of the EL1 state, while KVM ARM must switch the EL1 state from the VM to the host OS and then again from the host OS to the new VM. Finally, KVM ARM also has to disable and enable traps and Stage-2 translation on each transition, which Xen ARM does not have to do.

For the I/O Latency microbenchmarks, a surprising result is that Xen ARM is slower than KVM ARM in both directions. These microbenchmarks measure the time from when a network I/O event is initiated by a sender until the receiver is notified, not including additional time spent transferring data. I/O latency is an especially important metric for real-time sensitive operations and many networking applications. The key insight to understanding the results is to see that Xen ARM does not benefit from its faster VM-to-hypervisor transition mechanism in this case because Xen ARM must switch between two separate VMs, Dom0 and a DomU, to process network I/O. Type 1 hypervisors only implement a limited set of functionality in the hypervisor directly, namely scheduling, memory management, the interrupt controller, and timers for Xen ARM. All other functionality, for example network and storage drivers are implemented in the special privileged VM, Dom0. Therefore, a VM performing I/O has to communicate with Dom0 and not just the Xen hypervisor, which means not just trapping to EL2, but also going to EL1 to run Dom0.

I/O Latency Out is much worse on Xen ARM than KVM ARM. When KVM ARM sends a network packet, it traps to the hypervisor, context switching the EL1 state, and then the host OS instance directly sends the data on the physical network. Xen ARM, on the other hand, traps from the VM to the hypervisor, which then signals a different VM, Dom0, and Dom0 then sends the data on the physical network. This signaling between VMs on Xen is slow for two main reasons. First, because the VM and Dom0 run on different physical CPUs, Xen must send a physical IPI from the CPU running the VM to the CPU running Dom0. Second, Xen actually switches from Dom0 to a special VM, called the idle domain, when Dom0 is idling and waiting for I/O. Thus, when Xen signals Dom0 to perform I/O on behalf of a VM, it must perform a VM switch from the idle domain to Dom0. We verified that changing the configuration of Xen to pinning both the VM and Dom0 to the same physical CPU or not specifying any pinning resulted in similar or worse results than reported in Table II, so the qualitative results are not specific to our configuration.

It is interesting to note that KVM x86 is much faster than everything else on I/O Latency Out. KVM on both ARM and x86 involve the same control path of transitioning from the VM to the hypervisor. While the path is conceptually similar to half of the path for the Hypercall microbenchmark, the result for the I/O Latency Out microbenchmark is not 50% of the Hypercall cost on neither platform. The reason is that for KVM x86, transitioning from the VM to the hypervisor accounts for only about 40% of the Hypercall cost, while transitioning from the hypervisor to the VM is the majority of the cost (a few cycles are spent handling the noop hypercall in the hypervisor). On ARM, it is much more expensive to transition from the VM to the hypervisor than from the hypervisor to the VM, because reading back the VGIC state is expensive, as shown in Table III.

I/O Latency In behaves more similarly between Xen and KVM on ARM because they perform similar low-level operations. Xen traps from Dom0 running in EL1 to the hypervisor running in EL2 and signals the receiving VM, the reverse of the procedure described above, thereby sending a physical IPI and switching from the idle domain to the receiving VM in EL1. For KVM ARM, the Linux host OS receives the network packet via VHOST on a separate CPU, wakes up the receiving VM's VCPU thread to run on another CPU, thereby sending a physical IPI. The VCPU thread traps to EL2, switches the EL1 state from the host to the VM, then switches to the VM in EL1. The end result is that the cost is similar across both hypervisors, with KVM being slightly faster. While KVM ARM is slower on I/O Latency In than I/O Latency Out because it performs more work on the incoming path, Xen has similar performance on both Latency I/O In and Latency I/O Out because it performs similar low-level operations for both microbenchmarks.

V. Application Benchmark Results

We next ran a number of real application benchmark workloads to quantify how well the ARM virtualization extensions support different hypervisor software designs in the context of more realistic workloads. Table IV lists the

Kernbench	Compilation of the Linux 3.17.0 kernel using the allnoconfig for ARM using GCC 4.8.2.
Hackbench	`hackbench` [14] using Unix domain sockets and 100 process groups running with 500 loops.
SPECjvm2008	`SPECjvm2008` [15] 2008 benchmark running several real life applications and benchmarks specifically chosen to benchmark the performance of the Java Runtime Environment. We used 15.02 release of the Linaro AArch64 port of OpenJDK to run the the benchmark.
Netperf	`netperf` v2.6.0 starting netserver on the server and running with its 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` v2.3 on the remote client, which measures number of handled requests per second serving the 41 KB index file of the GCC 4.4 manual using 100 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 IV: Application Benchmarks

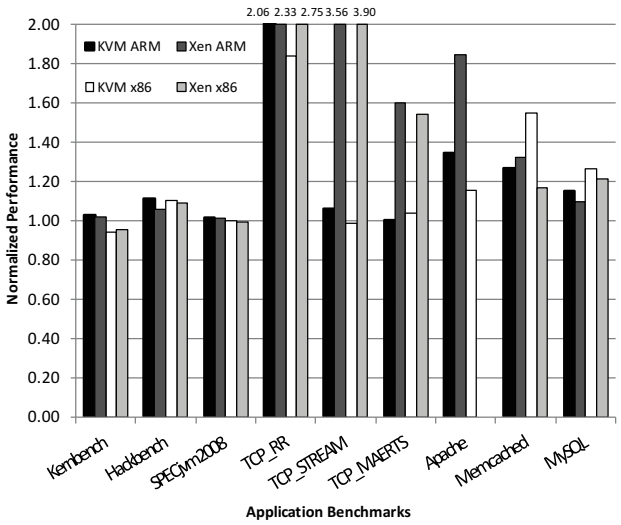

Figure 4: Application Benchmark Performance

application workloads we used, which include a mix of widely-used CPU and I/O intensive benchmark workloads. For workloads involving a client and a server, we ran the client on a dedicated machine and the server on the configuration being measured, ensuring that the client was never saturated during any of our experiments. We ran these workloads natively and on both KVM and Xen on both ARM and x86, the latter to provide a baseline comparison.

Given the differences in hardware platforms, our focus was not on measuring absolute performance [13], but rather the relative performance differences between virtualized and native execution on each platform. Figure 4 shows the performance overhead of KVM and Xen on ARM and x86 compared to native execution on the respective platform. All numbers are normalized to 1 for native performance, so that lower numbers represent better performance. Unfortunately, the Apache benchmark could not run on Xen x86 because it caused a kernel panic in Dom0. We tried several versions of Xen and Linux, but faced the same problem. We reported this to the Xen developer community, and learned that this may be a Mellanox network driver bug exposed by Xen's I/O model. We also reported the issue to the Mellanox driver maintainers, but did not arrive at a solution.

Figure 4 shows that the application performance on KVM and Xen on ARM and x86 is not well correlated with their respective microbenchmark performance shown in Table II. Xen ARM has by far the lowest VM-to-hypervisor transition costs and the best performance for most of the microbenchmarks, yet its performance lags behind KVM ARM on many of the application benchmarks. KVM ARM substantially outperforms Xen ARM on the various Netperf benchmarks, TCP_STREAM, TCP_MAERTS, and TCP_RR, as well as Apache and Memcached, and performs only slightly worse on the rest of the application benchmarks. Xen ARM also does generally worse than KVM x86. Clearly, the differences in microbenchmark performance do not result in the same differences in real application performance.

Xen ARM achieves its biggest performance gain versus KVM ARM on Hackbench. Hackbench involves running lots of threads that are sleeping and waking up, requiring frequent IPIs for rescheduling. Xen ARM performs virtual IPIs much faster than KVM ARM, roughly a factor of two. Despite this microbenchmark performance advantage on a workload that performs frequent virtual IPIs, the resulting difference in Hackbench performance overhead is small, only 5% of native performance. Overall, across CPU-intensive workloads such as Kernbench, Hackbench and SPECjvm2008, the performance differences among the different hypervisors across different architectures is small.

Figure 4 shows that the largest differences in performance are for the I/O-intensive workloads. We first take a closer look at the Netperf results. Netperf TCP_RR is an I/O latency benchmark, which sends a 1 byte packet from a client to the Netperf server running in the VM, and the Netperf server sends the packet back to the client, and the process is repeated for 10 seconds. For the Netperf TCP_RR benchmark, both hypervisors show high overhead compared to native performance, but Xen is noticeably worse than KVM. To understand why, we analyzed the behavior of TCP_RR in further detail by using tcpdump [16] to capture timestamps on incoming and outgoing packets at the data link layer. We modified Linux's timestamping function to use the ARM architected counter, and took further steps to ensure that the counter values were synchronized across all PCPUs, VMs, and the hypervisor. This allowed us to analyze the latency between operations happening in the VM and the host. Table V shows the detailed measurements.

Table V shows that the time per transaction increases significantly from 41.8 μs when running natively to 86.3 μs

	Native	KVM	Xen
Trans/s	23,911	11,591	10,253
Time/trans (μs)	41.8	86.3	97.5
Overhead (μs)	-	44.5	55.7
send to recv (μs)	29.7	29.8	33.9
recv to send (μs)	14.5	53.0	64.6
recv to VM recv (μs)	-	21.1	25.9
VM recv to VM send (μs)	-	16.9	17.4
VM send to send (μs)	-	15.0	21.4

Table V: Netperf TCP_RR Analysis on ARM

and 97.5 μs for KVM and Xen, respectively. The resulting overhead per transaction is 44.5 μs and 55.7 μs for KVM and Xen, respectively. To understand the source of this overhead, we decompose the time per transaction into separate steps. *send to recv* is the time between sending a packet from the physical server machine until a new response is received by the client, which is the time spent on the physical wire plus the client processing time. *recv to send* is the time spent at the physical server machine to receive a packet and send back a response, including potentially passing through the hypervisor and the VM in the virtualized configurations.

send to recv remains the same for KVM and native, because KVM does not interfere with normal Linux operations for sending or receiving network data. However, *send to recv* is slower on Xen, because the Xen hypervisor adds latency in handling incoming network packets. When a physical network packet arrives, the hardware raises an IRQ, which is handled in the Xen hypervisor, which translates the incoming physical IRQ to a virtual IRQ for Dom0, which runs the physical network device driver. However, since Dom0 is often idling when the network packet arrives, Xen must first switch from the idle domain to Dom0 before Dom0 can receive the incoming network packet, similar to the behavior of the I/O Latency benchmarks described in Section IV.

Since almost all the overhead is on the server for both KVM and Xen, we further decompose the *recv to send* time at the server into three components; the time from when the physical device driver receives the packet until it is delivered in the VM, *recv to VM recv*, the time from when the VM receives the packet until it sends a response, *VM recv to VM send*, and the time from when the VM delivers the response to the physical device driver, *VM send to send*. Table V shows that both KVM and Xen spend a similar amount of time receiving the packet inside the VM until being able to send a reply, and that this *VM recv to VM send* time is only slightly more time than the *recv to send* time spent when Netperf is running natively to process a packet. This suggests that the dominant overhead for both KVM and Xen is due to the time required by the hypervisor to process packets, the Linux host for KVM and Dom0 for Xen.

Table V also shows that Xen spends noticeably more time than KVM in delivering packets between the physical device driver and the VM. KVM only delays the packet on *recv to VM recv* and *VM send to send* by a total of 36.1 μs, where Xen delays the packet by 47.3 μs, an extra 11.2 μs. There are two main reasons why Xen performs worse. First, Xen's I/O latency is higher than KVM's as measured and explained by the I/O Latency In and Out microbenchmarks in Section IV. Second, Xen does not support zero-copy I/O, but instead must map a shared page between Dom0 and the VM using the Xen grant mechanism, and must copy data between the memory buffer used for DMA in Dom0 and the granted memory buffer from the VM. Each data copy incurs more than 3 μs of additional latency because of the complexities of establishing and utilizing the shared page via the grant mechanism across VMs, even though only a single byte of data needs to be copied.

Although Xen ARM can transition between the VM and hypervisor more quickly than KVM, Xen cannot utilize this advantage for the TCP_RR workload, because Xen must engage Dom0 to perform I/O on behalf of the VM, which results in several VM switches between idle domains and Dom0 or DomU, and because Xen must perform expensive page mapping operations to copy data between the VM and Dom0. This is a direct consequence of Xen's software architecture and I/O model based on domains and a strict I/O isolation policy. Xen ends up spending so much time communicating between the VM and Dom0 that it completely dwarfs its low Hypercall cost for the TCP_RR workload and ends up having more overhead than KVM ARM, due to Xen's software architecture and I/O model in particular.

The hypervisor software architecture is also a dominant factor in other aspects of the Netperf results. For the Netperf TCP_STREAM benchmark, KVM has almost no overhead for x86 and ARM while Xen has more than 250% overhead. The reason for this large difference in performance is again due to Xen's lack of zero-copy I/O support, in this case particularly on the network receive path. The Netperf TCP_STREAM benchmark sends large quantities of data from a client to the Netperf server in the VM. Xen's Dom0, running Linux with the physical network device driver, cannot configure the network device to DMA the data directly into guest buffers, because Dom0 does not have access to the VM's memory. When Xen receives data, it must configure the network device to DMA the data into a Dom0 kernel memory buffer, signal the VM for incoming data, let Xen configure a shared memory buffer, and finally copy the incoming data from the Dom0 kernel buffer into the virtual device's shared buffer. KVM, on the other hand, has full access to the VM's memory and maintains shared memory buffers in the Virtio rings [7], such that the network device can DMA the data directly into a guest-visible buffer, resulting in significantly less overhead.

Furthermore, previous work [17] and discussions with the Xen maintainers confirm that supporting zero copy on x86 is problematic for Xen given its I/O model because doing so requires signaling all physical CPUs to locally invalidate TLBs when removing grant table entries for shared pages,

which proved more expensive than simply copying the data [18]. As a result, previous efforts to support zero copy on Xen x86 were abandoned. Xen ARM lacks the same zero copy support because the Dom0 network backend driver uses the same code as on x86. Whether zero copy support for Xen can be implemented efficiently on ARM, which has hardware support for broadcast TLB invalidate requests across multiple PCPUs, remains to be investigated.

For the Netperf TCP_MAERTS benchmark, Xen also has substantially higher overhead than KVM. The benchmark measures the network transmit path from the VM, the converse of the TCP_STREAM benchmark which measured the network receive path to the VM. It turns out that the Xen performance problem is due to a regression in Linux introduced in Linux v4.0-rc1 in an attempt to fight bufferbloat, and has not yet been fixed beyond manually tuning the Linux TCP configuration in the guest OS [19]. We confirmed that using an earlier version of Linux or tuning the TCP configuration in the guest using sysfs significantly reduced the overhead of Xen on the TCP_MAERTS benchmark.

Other than the Netperf workloads, the application workloads with the highest overhead were Apache and Memcached. We found that the performance bottleneck for KVM and Xen on ARM was due to network interrupt processing and delivery of virtual interrupts. Delivery of virtual interrupts is more expensive than handling physical IRQs on bare-metal, because it requires switching from the VM to the hypervisor, injecting a virtual interrupt to the VM, then switching back to the VM. Additionally, Xen and KVM both handle all virtual interrupts using a single VCPU, which, combined with the additional virtual interrupt delivery cost, fully utilizes the underlying PCPU. We verified this by distributing virtual interrupts across multiple VCPUs, which causes performance overhead to drop on KVM from 35% to 14% on Apache and from 26% to 8% on Memcached, and on Xen from 84% to 16% on Apache and from 32% to 9% on Memcached. Furthermore, we ran the workload natively with all physical interrupts assigned to a single physical CPU, and observed the same native performance, experimentally verifying that delivering virtual interrupts is more expensive than handling physical interrupts.

In summary, while the VM-to-hypervisor transition cost for a Type 1 hypervisor like Xen is much lower on ARM than for a Type 2 hypervisor like KVM, this difference is not easily observed for the application workloads. The reason is that Type 1 hypervisors typically only support CPU, memory, and interrupt virtualization directly in the hypervisors. CPU and memory virtualization has been highly optimized directly in hardware and, ignoring one-time page fault costs at start up, is performed largely without the hypervisor's involvement. That leaves only interrupt virtualization, which is indeed much faster for Type 1 hypervisor on ARM, confirmed by the Interrupt Controller Trap and Virtual IPI microbenchmarks shown in Section IV. While this contributes to Xen's slightly better Hackbench performance, the resulting application performance benefit overall is modest.

However, when VMs perform I/O operations such as sending or receiving network data, Type 1 hypervisors like Xen typically offload such handling to separate VMs to avoid having to re-implement all device drivers for the supported hardware and to avoid running a full driver and emulation stack directly in the Type 1 hypervisor, which would significantly increase the Trusted Computing Base and increase the attack surface of the hypervisor. Switching to a different VM to perform I/O on behalf of the VM has very similar costs on ARM compared to a Type 2 hypervisor approach of switching to the host on KVM. Additionally, KVM on ARM benefits from the hypervisor having privileged access to all physical resources, including the VM's memory, and from being directly integrated with the host OS, allowing for optimized physical interrupt handling, scheduling, and processing paths in some situations.

Despite the inability of both KVM and Xen to leverage the potential fast path of trapping from a VM running in EL1 to the hypervisor in EL2 without the need to run additional hypervisor functionality in EL1, our measurements show that both KVM and Xen on ARM can provide virtualization overhead similar to, and in some cases better than, their respective x86 counterparts.

VI. Architecture Improvements

To make it possible for modern hypervisors to achieve low VM-to-hypervisor transition costs on real application workloads, some changes needed to be made to the ARM hardware virtualization support. Building on our experiences with the design, implementation, and performance measurement of KVM ARM and working in conjunction with ARM, a set of improvements have been made to bring the fast VM-to-hypervisor transition costs possible in limited circumstances with Type 1 hypervisors, to a broader range of application workloads when using Type 2 hypervisors. These improvements are the Virtualization Host Extensions (VHE), which are now part of a new revision of the ARM 64-bit architecture, ARMv8.1 [20]. VHE allows running an OS designed to run in EL1 to run in EL2 without substantial modifications to the OS source code. We show how this allows KVM ARM and its Linux host kernel to run entirely in EL2 without substantial modifications to Linux.

VHE is provided through the addition of a new control bit, the E2H bit, which is set at system boot when installing a Type 2 hypervisor that uses VHE. If the bit is not set, ARMv8.1 behaves the same as ARMv8 in terms of hardware virtualization support, preserving backwards compatibility with existing hypervisors. When the bit is set, VHE enables three main features.

First, VHE expands EL2, adding additional physical register state to the CPU, such that any register and functionality

available in EL1 is also available in EL2. For example, EL1 has two registers, TTBR0_EL1 and TTBR1_EL1, the first used to look up the page tables for virtual addresses (VAs) in the lower VA range, and the second in the upper VA range. This provides a convenient and efficient method for splitting the VA space between userspace and the kernel. However, without VHE, EL2 only has one page table base register, TTBR0_EL2, making it problematic to support the split VA space of EL1 when running in EL2. With VHE, EL2 gets a second page table base register, TTBR1_EL2, making it possible to support split VA space in EL2 in the same way as provided in EL1. This enables a Type 2 hypervisor integrated with a host OS to support a split VA space in EL2, which is necessary to run the host OS in EL2 so it can manage the VA space between userspace and the kernel.

Second, VHE provides a mechanism to access the extra EL2 register state transparently. Simply providing extra EL2 registers is not sufficient to run unmodified OSes in EL2, because existing OSes are written to access EL1 registers. For example, Linux is written to use TTBR1_EL1, which does not affect the translation system running in EL2. Providing the additional register TTBR1_EL2 would still require modifying Linux to use the TTBR1_EL2 instead of the TTBR1_EL1 when running in EL2 vs. EL1, respectively. To avoid forcing OS vendors to add this extra level of complexity to the software, VHE allows unmodified software to execute in EL2 and transparently access EL2 registers using the EL1 system register instruction encodings. For example, current OS software reads the TTBR1_EL1 register with the instruction `mrs x1, ttbr1_el1`. With VHE, the software still executes the same instruction, but the hardware actually accesses the TTBR1_EL2 register. As long as the E2H bit is set, accesses to EL1 registers performed in EL2 actually access EL2 registers, transparently rewriting register accesses to EL2, as described above. A new set of special instructions are added to access the EL1 registers in EL2, which the hypervisor can use to switch between VMs, which will run in EL1. For example, if the hypervisor wishes to access the guest's TTBR1_EL1, it will use the instruction `mrs x1, ttb1_el21`.

Third, VHE expands the memory translation capabilities of EL2. In ARMv8, EL2 and EL1 use different page table formats so that software written to run in EL1 must be modified to run in EL2. In ARMv8.1, the EL2 page table format is now compatible with the EL1 format when the E2H bit is set. As a result, an OS that was previously run in EL1 can now run in EL2 without being modified because it can use the same EL1 page table format.

Figure 5 shows how Type 1 and Type 2 hypervisors map to the architecture with VHE. Type 1 hypervisors do not set the E2H bit introduced with VHE, and EL2 behaves exactly as in ARMv8 and described in Section II. Type 2 hypervisors set the E2H bit when the system boots, and the host OS kernel runs exclusively in EL2, and never in EL1. The Type 2

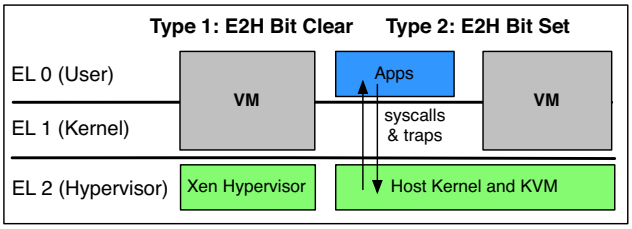

Figure 5: Virtualization Host Extensions (VHE)

hypervisor kernel can run unmodified in EL2, because VHE provides an equivalent EL2 register for every EL1 register and transparently rewrites EL1 register accesses from EL2 to EL2 register accesses, and because the page table formats between EL1 and EL2 are now compatible. Transitions from host userspace to host kernel happen directly from EL0 to EL2, for example to handle a system call, as indicated by the arrows in Figure 5. Transitions from the VM to the hypervisor now happen without having to context switch EL1 state, because EL1 is not used by the hypervisor.

ARMv8.1 differs from the x86 approach in two key ways. First, ARMv8.1 introduces more additional hardware state so that a VM running in EL1 does not need to save a substantial amount of state before switching to running the hypervisor in EL2 because the EL2 state is separate and backed by additional hardware registers. This minimizes the cost of VM-to-hypervisor transitions because trapping from EL1 to EL2 does not require saving and restoring state beyond general purpose registers to and from memory. In contrast, recall that the x86 approach adds CPU virtualization support by adding root and non-root mode as orthogonal concepts from the CPU privilege modes, but does not introduce additional hardware register state like ARM. As a result, switching between root and non-root modes requires transferring state between hardware registers and memory. The cost of this is ameliorated by implementing the state transfer in hardware, but while this avoids the need to do additional instruction fetch and decode, accessing memory is still expected to be more expensive than having extra hardware register state. Second, ARMv8.1 preserves the RISC-style approach of allowing software more fine-grained control over which state needs to be switched for which purposes instead of fixing this in hardware, potentially making it possible to build hypervisors with lower overhead, compared to x86.

A Type 2 hypervisor originally designed for ARMv8 must be modified to benefit from VHE. A patch set has been developed to add VHE support to KVM ARM. This involves rewriting parts of the code to allow run-time adaptations of the hypervisor, such that the same kernel binary can run on both legacy ARMv8 hardware and benefit from VHE-enabled ARMv8.1 hardware. The code to support VHE has been developed using ARM software models as ARMv8.1 hardware is not yet available. We were therefore not able to evaluate the performance of KVM ARM using VHE, but our

findings in Sections IV and V show that this addition to the hardware design could have a noticeable positive effect on KVM ARM performance, potentially improving Hypercall and I/O Latency Out performance by more than an order of magnitude, improving more realistic I/O workloads by 10% to 20%, and yielding superior performance to a Type 1 hypervisor such as Xen which must still rely on Dom0 running in EL1 for I/O operations.

VII. Related Work

Virtualization goes back to the 1970s [8], but saw a resurgence in the 2000s with the emergence of x86 hypervisors and later x86 hardware virtualization support [5], [6], [4]. Much work has been done on analyzing and improving the performance of x86 virtualization [21], [17], [22], [23], [24], [25], [26], [27], [28]. While some techniques such as nested page tables have made their way from x86 to ARM, much of the x86 virtualization work has limited applicability to ARM for two reasons. First, earlier work focused on techniques to overcome the absence of x86 hardware virtualization support. For example, studies of paravirtualized VM performance [17] are not directly applicable to systems optimized with hardware virtualization support.

Second, later work based on x86 hardware virtualization support leverages hardware features that are in many cases substantially different from ARM. For example, ELI [24] reduces the overhead of device passthrough I/O coming from interrupt processing by applying an x86-specific technique to directly deliver physical interrupts to VMs. This technique does not work on ARM, as ARM does not use Interrupt Descriptor Tables (IDTs), but instead reads the interrupt number from a single hardware register and performs lookups of interrupt service routines from a strictly software-managed table. In contrast, our work focuses on ARM-specific hardware virtualization support and its performance on modern hypervisors running multiprocessor VMs.

Full-system virtualization of the ARM architecture is a relatively unexplored research area. Early approaches were software only, could not run unmodified guest OSes, and often suffered from poor performance [29], [30], [31], [32]. More recent approaches leverage ARM hardware virtualization support. The earliest study of ARM hardware virtualization support was based on a software simulator and a simple hypervisor without SMP support, but due to the lack of hardware or a cycle-accurate simulator, no real performance evaluation was possible [33].

KVM ARM was the first hypervisor to use ARM hardware virtualization support to run unmodified guest OSes on multi-core hardware [2], [34]. We expand on our previous work by (1) measuring virtualization performance on ARM server hardware for the first time, (2) providing the first performance comparison between KVM and Xen on both ARM and x86, (3) quantifying the true cost of split-mode virtualization due to the need to save and restore more state to memory when transitioning from a VM to the hypervisor compared to Type 1 hypervisors on ARM, and (4) identifying the root causes of overhead for KVM and Xen on ARM for real application workloads including those involving network I/O.

VIII. Conclusions

We present the first study of ARM virtualization performance on server hardware, including multi-core measurements of the two main ARM hypervisors, KVM and Xen. We introduce a suite of microbenchmarks to measure common hypervisor operations on multi-core systems. Using this suite, we show that ARM enables Type 1 hypervisors such as Xen to transition between a VM and the hypervisor much faster than on x86, but that this low transition cost does not extend to Type 2 hypervisors such as KVM because they cannot run entirely in the EL2 CPU mode ARM designed for running hypervisors. While this fast transition cost is useful for supporting virtual interrupts, it does not help with I/O performance because a Type 1 hypervisor like Xen has to communicate with I/O backends in a special Dom0 VM, requiring more complex interactions than simply transitioning to and from the EL2 CPU mode.

We show that current hypervisor designs cannot leverage ARM's potentially fast VM-to-hypervisor transition cost in practice for real application workloads. KVM ARM actually exceeds the performance of Xen ARM for most real application workloads involving I/O. This is due to differences in hypervisor software design and implementation that play a larger role than how the hardware supports low-level hypervisor operations. For example, KVM ARM easily provides zero copy I/O because its host OS has full access to all of the VM's memory, where Xen enforces a strict I/O isolation policy resulting in poor performance despite Xen's much faster VM-to-hypervisor transition mechanism. We show that ARM hypervisors have similar overhead to their x86 counterparts on real applications. Finally, we show how improvements to the ARM architecture may allow Type 2 hypervisors to bring ARM's fast VM-to-hypervisor transition cost to real application workloads involving I/O.

Acknowledgments

Marc Zyngier provided insights and implemented large parts of KVM ARM. Eric Auger helped add VHOST support to KVM ARM. Peter Maydell helped on QEMU internals and configuration. Ian Campbell and Stefano Stabellini helped on Xen internals and in developing our measurement frameworks for Xen ARM. Leigh Stoller, Mike Hibler, and Robert Ricci provided timely support for CloudLab. Martha Kim and Simha Sethumadhavan gave feedback on earlier drafts of this paper. This work was supported in part by Huawei Technologies, a Google Research Award, and NSF grants CNS-1162447, CNS-1422909, and CCF-1162021.

REFERENCES

[1] ARM Ltd., "ARM architecture reference manual ARMv8-A DDI0487A.a," Sep. 2013.

[2] C. Dall and J. Nieh, "KVM/ARM: The design and implementation of the Linux ARM hypervisor," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2014, pp. 333–348.

[3] "Xen ARM with virtualization extensions," http://wiki.xen.org/wiki/Xen_ARM_with_Virtualization_Extensions.

[4] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "**kvm**: The Linux virtual machine monitor," in *Proceedings of the Ottawa Linux Symposium*, vol. 1, Jun. 2007, pp. 225–230.

[5] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang, "Bringing virtualization to the x86 architecture with the original VMware workstation," *ACM Transactions on Computer Systems*, vol. 30, no. 4, pp. 12:1–12:51, Nov. 2012.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the 19th Symposium on Operating Systems Principles*, Oct. 2003, pp. 164–177.

[7] R. Russell, "virtio: Towards a de-facto standard for virtual I/O devices," *SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 95–103, Jul. 2008.

[8] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, Jul. 1974.

[9] "CloudLab," http://www.cloudlab.us.

[10] Hewlett-Packard, "HP Moonshot-45XGc switch module," http://www8.hp.com/us/en/products/moonshot-systems/product-detail.html?oid=7398915.

[11] "Tuning xen for performance," http://wiki.xen.org/wiki/Tuning_Xen_for_Performance, accessed: Jul. 2015.

[12] Intel Corporation, "Intel 64 and IA-32 architectures software developer's manual, 325384-056US," Sep. 2015.

[13] C. Dall, S.-W. Li, J. T. Lim, and J. Nieh, "A measurement study of ARM virtualization performance," Department of Computer Science, Columbia University, Tech. Rep. CUCS-021-15, Nov. 2015.

[14] "Hackbench," Jan. 2008, http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c.

[15] "Specjvm2008," https://www.spec.org/jvm2008.

[16] "Tcpdump," http://www.tcpdump.org/tcpdump_man.html.

[17] J. R. Santos, Y. Turner, G. J. Janakiraman, and I. Pratt, "Bridging the gap between software and hardware techniques for I/O virtualization," in *Proceedings of the 2008 USENIX Annual Technical Conference*, Jun. 2008, pp. 29–42.

[18] Ian Campbell, "Personal communication," Apr. 2015.

[19] Linux ARM Kernel Mailing List, ""tcp: refine TSO autosizing" causes performance regression on Xen," Apr. 2015, http://lists.infradead.org/pipermail/linux-arm-kernel/2015-April/336497.html.

[20] D. Brash, "The ARMv8-A architecture and its ongoing development," Dec. 2014, http://community.arm.com/groups/processors/blog/2014/12/02/the-armv8-a-architecture-and-its-ongoing-development.

[21] J. Sugerman, G. Venkitachalam, and B.-H. Lim, "Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor," in *Proceedings of the 2001 USENIX Annual Technical Conference*, Jun. 2001, pp. 1–14.

[22] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006, pp. 2–13.

[23] O. Agesen, J. Mattson, R. Rugina, and J. Sheldon, "Software techniques for avoiding hardware virtualization exits," in *Proceedings of the 2012 USENIX Annual Technical Conference*, Jun. 2012, pp. 373–385.

[24] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir, "ELI: Bare-metal performance for I/O virtualization," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2012, pp. 411–422.

[25] L. Cherkasova and R. Gardner, "Measuring CPU overhead for I/O processing in the Xen virtual machine monitor." in *Proceedings of the 2005 USENIX Annual Technical Conference*, Jun. 2005, pp. 387–390.

[26] K. Gamage and X. Kompella, "Opportunistic flooding to improve tcp transmit performance in virtualized clouds." in *Proceedings of the 2nd Symposium on Cloud Computing*, 2011, pp. 24:1–24:14.

[27] J. Heo and R. Taheri, "Virtualizing latency-sensitive applications: Where does the overhead come from?" *VMware Technical Journal*, vol. 2, no. 2, pp. 21–30, Dec. 2013.

[28] J. Buell, D. Hecht, J. Heo, K. Saladi, and H. R. Taheri, "Methodology for performance analysis of VMware vSphere under tier-1 applications," *VMware Technical Journal*, vol. 2, no. 1, pp. 19–28, Jun. 2013.

[29] J. Hwang, S. Suh, S. Heo, C. Park, J. Ryu, S. Park, and C. Kim, "Xen on ARM: System virtualization using Xen hypervisor for ARM-based secure mobile phones," in *Proceedings of the 5th Consumer Communications and Newtork Conference*, Jan. 2008, pp. 257–261.

[30] C. Dall and J. Nieh, "KVM for ARM," in *Proceedings of the Ottawa Linux Symposium*, Jul. 2010, pp. 45–56.

[31] J.-H. Ding, C.-J. Lin, P.-H. Chang, C.-H. Tsang, W.-C. Hsu, and Y.-C. Chung, "ARMvisor: System virtualization for ARM," in *Proceedings of the Ottawa Linux Symposium*, Jul. 2012, pp. 93–107.

[32] K. Barr, P. Bungale, S. Deasy, V. Gyuris, P. Hung, C. Newell, H. Tuch, and B. Zoppis, "The VMware mobile virtualization platform: Is that a hypervisor in your pocket?" *SIGOPS Operating Systems Review*, vol. 44, no. 4, pp. 124–135, Dec. 2010.

[33] P. Varanasi and G. Heiser, "Hardware-supported virtualization on ARM," in *Proceedings of the 2nd Asia-Pacific Workshop on Systems*, Jul. 2011, pp. 11:1–11:5.

[34] C. Dall and J. Nieh, "KVM/ARM: Experiences building the Linux ARM hypervisor," Department of Computer Science, Columbia University, Tech. Rep. CUCS-010-13, Apr. 2013.