Optimizing the Design and Implementation of the Linux ARM Hypervisor

Christoffer Dall Shih-Wei Li Jason Nieh
Department of Computer Science
Columbia University
{*cdall,shihwei,nieh*}*@cs.columbia.edu*

Abstract

Modern hypervisor designs for both ARM and x86 virtualization rely on running an operating system kernel, the hypervisor OS kernel, to support hypervisor functionality. While x86 hypervisors effectively leverage architectural support to run the kernel, existing ARM hypervisors map poorly to the virtualization features of the ARM architecture, resulting in worse performance. We identify the key reason for this problem is the need to multiplex kernel mode state between the hypervisor and virtual machines, which each run their own kernel. To address this problem, we take a fundamentally different approach to hypervisor design that runs the hypervisor together with its OS kernel in a separate CPU mode from kernel mode. Using this approach, we redesign KVM/ARM to leverage a separate ARM CPU mode for running both the hypervisor and its OS kernel. We show what changes are required in Linux to implement this on current ARM hardware as well as how newer ARM architectural support can be used to support this approach without any changes to Linux other than to KVM/ARM itself. We show that our redesign and optimizations can result in an order of magnitude performance improvement for KVM/ARM, and can provide faster performance than x86 on key hypervisor operations. As a result, many aspects of our design have been successfully merged into mainline Linux.

1 Introduction

Given their customizability and power efficiency, ARM CPUs have become an attractive option across a wide range of computer systems, from their dominance in mobile and embedded systems to their increasing popularity in server systems. Recognizing that virtualization is a key technology for the successful deployment of ARM hardware, modern ARM CPUs include hardware support for virtualization, the Virtualization Extensions (VE). Popular ARM hypervisors, including KVM [14] and Xen [30], utilize VE to run unmodified commodity operating systems (OSes) and applications across a wide range of deployment scenarios for virtualization, from enterprise servers to locomotive computer systems [5]. Despite these successes, we have shown that ARM virtualization costs remain too high for important deployment scenarios, including network-intensive workloads such as network functions virtualization (NFV) [14, 12].

Hypervisor designs for ARM and x86 virtualization rely on running a full OS kernel to support the hypervisor functionality. This is true for both Type 1 hypervisors which run an isolated hypervisor runtime and Type 2 hypervisors which integrate with a host OS [17]. KVM, a Type 2 hypervisor, is integrated with the Linux kernel and leverages the Linux kernel for common OS functionality such as scheduling, memory management, and hardware support. Similarly, Xen, a Type 1 hypervisor, runs a full copy of Linux in a special privileged Virtual Machine (VM) called Dom0 to leverage existing Linux drivers to provide I/O for other VMs. These *hypervisor OS kernels* which support the hypervisor run in the CPU's kernel mode just like OS kernels run when not using virtualization. Modern hypervisors use hardware support for virtualization, avoiding the need to deprivilege the guest OS kernel in a VM to run in user mode [8]. As each VM runs a guest OS kernel in addition to the hypervisor OS kernel, and both kernels run in the same kernel mode, the shared hardware state belonging to kernel mode is multiplexed among the OS kernels. When a VM is running on the CPU, the VM's guest OS kernel is using the CPU's kernel mode, but when it becomes necessary to run the hypervisor, for example to perform I/O on behalf of the VM, the hypervisor OS kernel takes over using the CPU's kernel mode.

Transitioning from the guest OS kernel to the hypervisor OS kernel involves saving the guest kernel's state and restoring the hypervisor kernel's state, and vice versa. This save and restore operation is necessary because both the guest and hypervisor OS kernels use the same hardware state such as registers and configuration settings, but in different contexts. On x86, these transitions happen using operations architecturally defined as part of the Intel Virtual Machine Extensions (VMX). These hardware operations save and

restore the entire kernel mode register state, typically as a result of executing a single instruction. Unlike x86, ARM does not provide a hardware mechanism to save and restore kernel mode state, but instead relies on software performing these operations on each register, which results in much higher overhead. The cost of transitioning from a VM to the hypervisor can be many times worse on ARM than x86 [12].

To address this problem, we present a new hypervisor design and implementation that takes advantage of unique features of the ARM architectural support for virtualization in the context of Type 2 hypervisors. We take a fundamentally different approach that runs the hypervisor together with its OS kernel in a separate CPU mode from kernel mode. ARM VE provides an extra hypervisor CPU mode, EL2, designed to run standalone hypervisors. EL2 is a separate mode from the EL1 kernel mode, and the architecture allows switching from EL1 to EL2 without saving or restoring any EL1 register state. In this design, the hypervisor and its OS kernel no longer run in EL1, but EL1 is reserved exclusively to be used by VMs. This means that the kernel mode hardware state no longer has to be multiplexed between the hypervisor OS kernel and a VM's guest OS kernel, and transitioning between the two does not require saving and restoring any kernel mode state. This new design, using separate hardware state for VMs and the hypervisor OS kernel can significantly improve hypervisor performance.

Our new hypervisor design benefits from the Virtualization Host Extensions (VHE) introduced in ARMv8.1. With VHE, our design does not require any changes to existing hypervisor OS kernels. Without VHE, our design requires modifications to the hypervisor OS kernel so it can run in EL2 instead of EL1. Although Type 1 hypervisors also suffer from poor performance due to slow transitions between the hypervisor OS kernel and guest OS kernels, our design is not easily applicable to Type 1 hypervisors. We focus on improving the performance of Type 2 hypervisors on ARM given their widespread popularity, which is at least in part due to their benefits over Type 1 hypervisors on ARM. ARM hardware does not have the same legacy and standards as x86, so Type 1 hypervisors have to be manually ported to every hardware platform they support. Type 2 hypervisors leverage their host OS and are automatically supported on all hardware platforms supported by their host OS.

Running the hypervisor and its OS kernel in a separate CPU mode with its own hardware state allows a number of improvements to the hypervisor implementation. First, transitioning from the VM to the hypervisor no longer requires saving and restoring the kernel mode register state. Second, the hypervisor OS kernel can program hardware state used by the VM directly when needed, avoiding extra copying to and from intermediate data strutures. Third, the hypervisor and its OS kernel no longer need to operate across different CPU modes with separate address spaces which requires separate data structures and duplicated code. Instead, the hypervisor can directly leverage existing OS kernel functionality while at the same time configure ARM hardware virtualization features, leading to reduced code complexity and improved performance.

We have implemented our approach by redesigning KVM/ARM and demonstrated that it is effective at providing significant performance benefits with reduced implementation complexity. A number of our changes have been merged into mainline Linux over the course of Linux kernel versions v4.5 through v4.8, with additional changes scheduled to be applied in upcoming kernel versions. We show that our redesign and optimizations can result in an order of magnitude performance improvement for KVM/ARM in microbenchmarks, and can reduce virtualization overhead by more than 50% for real application workloads. We show that both hardware and software need to work together to provide the optimal performance. We also show that our optimized KVM/ARM provides significant performance gains compared to x86, indicating that our hypervisor design combined with the required architectural support for virtualization provides a superior approach to x86 hardware virtualization.

2 Background

We first provide a brief overview of current state-of-the-art Type 2 hypervisor designs on both x86 and ARM and discuss how they must multiplex kernel mode to run both their VM and hypervisor OS kernels using hardware virtualization support. For architectural support for virtualization on x86, we focus on Intel VMX, though AMD-V is similar for the purposes of this discussion.

2.1 Intel VMX

The Intel Virtual Machine Extensions (VMX) [21], support running VMs through the addition of a new feature, *VMX operations*. When VMX is enabled, the CPU can be in one of two VMX operations, VMX root or VMX non-root operation. Root operation allows full control of the hardware and is for running the hypervisor. Non-root operation is restricted to operate only on virtual hardware and is for running VMs. VMX provides memory virtualization through Extended Page Tables (EPT) which limits the memory the VM can access in VMX non-root. Both VMX root and non-root operation have the same full set of CPU modes available to them, including both user and kernel mode, but certain sensitive instructions executed in non-root operation cause a transition to root operation to allow the hypervisor to maintain complete control of the system. The hypervisor OS kernel runs in root operation and a VM's guest OS kernel runs in non-root operation, but both run in the same CPU mode. Since the hypervisor and the VM have separate execution contexts in form of register state and configuration state, all of this state must be multiplexed between root and non-root operation.

VMX supports this multiplexing in hardware by defining two VMX transitions, VM Entry and VM Exit. VM Entry transitions from root to non-root operation which happens when the hypervisor decides to run a VM by executing a specific instruction. VM Exit transitions from non-root to root operation which transfers control back to the hypervisor on certain events such as hardware interrupts or when the VM attempts to perform I/O or access sensitive state. The transitions are managed by hardware using an in-memory data structure called the Virtual-Machine Control Structure (VMCS). VMX root and non-root operation do not have separate CPU hardware modes, but VMX instead multiplexes the modes between the hypervisor and VM by saving and restoring CPU state to memory using hardware VMX transitions.

2.2 ARM VE

ARM took a very different approach than x86 in adding hardware virtualization support. Instead of introducing an orthogonal feature to distinguish between the hypervisor and VM operation, ARM extended the existing CPU mode hierarchy, originally just EL0 user mode and EL1 kernel mode, by adding a separate more privileged mode called EL2 to run the hypervisor. Although ARM refers to EL0, EL1, and EL2 as exception levels, we refer to them here as CPU modes to simplify the discussion. EL2 cannot be used to run existing unmodified OS kernels for a number of reasons. For example, EL2 has its own set of control registers and has a limited and separate address space compared to EL1, so it is not compatible with EL1. Furthermore, EL2 does not easily support running userspace applications in EL0 which expect to interact with a kernel running in EL1 instead of EL2.

Therefore, both the hypervisor and VM OS kernels must run in EL1, and this mode must be multiplexed between the two execution contexts. On ARM, this can be done by software running in EL2. EL2 is a strictly more privileged mode than user and kernel modes, EL0 and EL1, respectively, and EL2 has its own execution context defined by register and control state, and can therefore completely switch the execution context of both EL0 and EL1 in software, similar to how the kernel in EL1 context switches between multiple userspace processes running in EL0.

When both the hypervisor and VM OS kernels run at the same privilege level on ARM without an equivalent feature to x86 VMX operations, an obvious question is how to differentiate between the roles of the hypervisor and the VM kernel. The hypervisor kernel should be in full control of the underlying physical hardware, while the VM kernel should be limited to the control of virtual hardware resources. This can be accomplished by using ARM VE which allows fine grained control of the capabilities of EL1. Software running in EL2 can enable certain sensitive instructions and events executed in EL0 or EL1 to trap to EL2. For example, similar to x86 EPT, ARM VE provides memory virtualization by adding an additional stage of address translation, the stage 2 translations. Stage 2 translations are controlled from EL2 and only affect software executing in EL1 and EL0. Hypervisor software running in EL2 can therefore completely disable the stage 2 translations when running the hypervisor OS kernel, giving it full access to all physical memory on the system, and conversely enable stage 2 translations when running VM kernels to limit VMs to manage memory allocated to them.

ARM VE supports the multiplexing of EL1 analogously to how EL0 is multiplexed between processes using EL1. Because EL2 is a separate and strictly more privileged mode than EL1, hypervisor software in EL2 can multiplex the entire EL1 state by saving and restoring each register and configuration state, one by one, to and from memory. In line with the RISC design of ARM, and in contrast to the CISC design of x86, ARM does not provide any hardware mechanism to multiplex EL1 between the hypervisor and VM kernels, but instead relies on existing simpler mechanisms in the architecture. For example, if a VM kernel tries to halt the physical processor, because this is a sensitive instruction and the VM is not allowed to control the physical CPU resource, this instruction will cause a trap to the more privileged EL2 mode, which can then reuse existing instructions to save and restore state and switch the EL1 execution context to the hypervisor kernel context, configure EL1 to have full access to the hardware, and return to EL1 to run the hypervisor OS kernel.

2.3 KVM

Figure 1 compares how the KVM hypervisor runs using x86 VMX versus ARM VE. We refer to the hypervisor OS kernel as the host OS kernel, the more commonly used term with KVM, and applications interacting directly with the OS, and running outside of a VM, as host user space. Figure 1(a) shows how KVM x86 works. The hypervisor and host OS run in root operation, with the host user space running in the least privileged CPU mode level 3, and the host kernel running in the privileged CPU mode, level 0, similar to running on a native system. All of the VM runs in non-root operation and the VM user space and kernel also run in level 3 and level 0, respectively. Transitions between root and non-root mode are done in hardware using the atomic VMX transitions, VM Entry and VM Exit.

Figure 1(b) shows how KVM/ARM works. Since the host OS kernel cannot run in EL2, but EL2 is needed to enable the virtualization features and to multiplex EL1, KVM/ARM uses split-mode virtualization [14] to support both the host OS kernel running in EL1 and at the same time run software in EL2 to manage the virtualization features and multiplex EL1. Most of the hypervisor functionality runs in EL1 with full access to the hardware as part of the host OS kernel, and a small layer, the *lowvisor*, runs in EL2.

When KVM x86 runs a VM, it issues a single instruction to perform the VM Entry. The VM Entry operation saves the

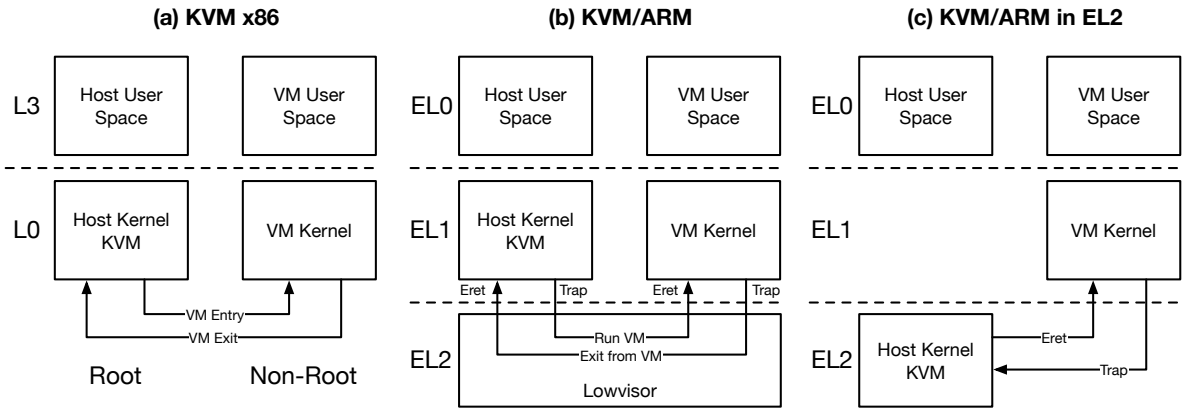

Figure 1: Hypervisor Designs and CPU Privilege Levels

hypervisor execution context of the processor to the VMCS and restores the VM execution context from the VMCS. On a VM Exit, x86 VMX performs the reverse operation and returns to the hypervisor. Since ARM does not have a single hardware mechanism to save and restore the entire state of the CPU, KVM/ARM issues a hypercall to trap to the lowvisor in EL2, which saves and restores all the registers and configuration state of the CPU, one by one, using a software defined structure in memory. After changing the EL0 and EL1 execution context to the VM, the lowvisor performs an *exception return* (eret) to the VM. When the VM traps to EL2, the lowvisor again saves and restores the entire state of the CPU and switches the execution context of EL0 and EL1 back to the hypervisor. As we shall see in Section 5.1, while the x86 VMX transitions are very complicated hardware operations, and the traps on ARM from EL1 to EL2 are cheap, multiplexing the kernel mode between two contexts ends up being much more expensive on ARM as a result of having to save and restore the entire CPU state in software.

3 Hypervisor OS Kernel Support

Running the hypervisor OS kernel in the same CPU mode as the VM kernels invariably results in multiplexing the kernel CPU mode, either in hardware or software, which adds overhead from the need to save and restore state. If instead a dedicated separate CPU mode were available to run the hypervisor OS kernel, this would avoid the need to multiplex a single mode and allow the hardware to simply trap from the VM to the hypervisor OS kernel to manage the underlying hardware and service the VM. Being able to transition back and forth between the full hypervisor functionality and the VM quickly without repeatedly saving and restoring the entire CPU state can reduce latency and improve virtualization performance.

Running the hypervisor OS kernel in a separate mode requires support from both hardware and software. The hardware must obviously be designed with a separate mode in addition to the mode used to run the VM kernel and VM user space. The hardware for the separate mode must support running full OS kernels that interact with user space applications. Furthermore, the hypervisor software must be designed to take advantage of running the hypervisor OS kernel in a separate CPU mode. As explained in Section 2, x86 does not meet these requirements because it does not have a separate CPU mode for the hypervisor OS kernel. ARM at least provides a separate CPU mode, EL2, but it was not designed for running hypervisor OS kernels. We show how this limitation can be overcome.

Figure 1(c) shows how KVM/ARM can be re-designed to run both the hypervisor (KVM) and its hypervisor OS kernel (Linux) together in EL2. This design is superior to previous ARM hypervisor designs including existing KVM/ARM and Xen on ARM, because it allows for very fast transitions between the VM and the hypervisor, including when running the hypervisor OS kernel, because there is no need to repeatedly save and restore the entire CPU state when transitioning between the VM and the hypervisor OS kernel. Furthermore, because the hypervisor is integrated with its hypervisor OS kernel, it can directly manage the underlying hardware using existing functionality such as device drivers in the hypervisor OS kernel without having to run special privileged VMs as is the case on Xen [12].

However, running an existing OS kernel in EL2 requires modifying the hardware or OS kernel, because EL2 was designed only to run hypervisors and lacks key features available in EL1, ARM's kernel mode, used to support OS kernels. First, EL2 uses a separate set of control registers accessed using different instructions than the EL1 control registers, causing incompabilities with a kernel written to run in EL1. Second, EL2 lacks support for host user space, which is needed to run applications such as QEMU, which provides device emulation. Running host user space applications in EL0 in conjunction with software running in EL2 without using EL1, as shown in Figure 1(c), requires handling exceptions from EL0 directly to EL2, for example to handle system calls, hardware interrupts, and page faults.

EL2 provides a *Trap General Exceptions* (TGE) bit to configure the CPU to route all exceptions from EL0 directly to EL2, but setting this bit also disables the use of virtual memory in EL0, which is problematic for real applications. Finally, EL2 uses a different page table format and only supports a single virtual address range, causing problems for a kernel written to use EL1's page table format and EL1's support for two separate virtual address space ranges.

3.1 Virtualization Host Extensions

To run existing hypervisor OS kernels in EL2 with almost no modifications, ARM introduced the Virtualization Host Extensions (VHE) in ARMv8.1. VHE is an architectural hardware modification that provides improved support for Type 2 hypervisors on ARM. It provides three key features.

First, VHE introduces additional EL2 registers to provide the same functionality available in EL1 to software running in EL2. VHE adds new virtual memory configuration registers, a new context ID register used for debugging, and a number of new registers to support a new timer. With these new registers in place, there is a corresponding EL2 system register for each EL1 system register. VHE then transparently changes the operation of instructions that normally access EL1 system registers to access EL2 registers instead when they run in EL2. By transparently changing the operation of the instructions, existing unmodified OSes written to issue EL1 system register instructions will instead access EL2 system registers when run in EL2. VHE also changes the bit layout of some EL2 system registers to share the same layout and semantics as their EL1 counterparts.

Second, VHE supports running host user space applications that use virtual memory in EL0 and interact directly with a kernel running in EL2. VHE introduces new functionality so that the EL0 virtual memory configuration can be managed by either EL1 or EL2, depending on a run time configuration setting, which allows EL2 to route exceptions from EL0 directly to EL2 and at the same time support virtual memory in EL0. VHE extends the functionality of the TGE bit such that when enabled and exceptions from EL0 are routed to EL2, virtual memory support is enabled in EL0 and controlled using EL2 page table registers. A Type 2 hypervisor will typically configure EL0 to use the EL2 system registers when running the hypervisor, and configure EL0 to use the EL1 system registers when running the VM.

Third, VHE changes the page table format of EL2 to use the same format as used in EL1, which avoids the need to change an existing OS kernel's page table management code to support different formats. VHE also adds support to EL2 for an additional separate virtual address space which can be used to provide the same split between kernel and user space addresses commonly used by existing ARM OSes in EL1 and EL0.

Using VHE to run Linux as the hypervisor OS kernel in conjunction with KVM requires very little effort. The early boot code in Linux simply sets a single bit in a register to enable VHE, and the kernel itself runs without further modification in EL2.

While the hypervisor OS kernel can run largely unmodified in EL2, the hypervisor itself must be modified to run with VHE. In particular, because EL1 system register access instructions are changed to access EL2 registers instead, the hypervisor needs an alternative mechanism to access the real EL1 registers, for example to prepare a VM's execution context. For this purpose, VHE adds new instructions, the `_EL12` instructions, which access EL1 registers when running in EL2 with VHE enabled. The hypervisor must be modified to replace all EL1 access instructions that should continue to access EL1 registers with the new `_EL12` access instructions when using VHE, and use the original EL1 access instructions when running without VHE.

3.2 *el2*Linux

Unfortunately, VHE hardware is not yet publicly available and remains an optional extension to the ARM architecture. As an alternative, we introduce *el2*Linux [11], a lightly modified version of Linux that runs in EL2 on non-VHE hardware. *el2*Linux brings the benefits of running Linux as the hypervisor OS kernel in a separate CPU mode to existing hardware alongside the KVM hypervisor. It involves three main kernel modifications to Linux.

First, to control its own CPU mode, Linux must access EL2 register state when running in EL2, and we modify the Linux kernel source code as needed to access EL2 system registers instead of EL1 registers. This can be done using either build time conditionals or at runtime using instruction patching to avoid overhead from introducing additional conditional code paths in the kernel.

Second, to support host user space applications such as QEMU in EL0 interacting with a kernel running in EL2, we install a tiny runtime in EL1, which includes an exception vector to forward exceptions to EL2 by issuing a hypercall instruction. The result is that exceptions from EL0 are forwarded to EL2 via EL1. However, this introduces two sources of additional overhead for applications running outside of a VM. One is a small overhead from going through EL1 to EL2 when handling an exception in EL0. The other is a larger overhead due to the need to multiplex EL1 between the EL1 runtime and a VM's guest OS kernel. While saving and restoring the EL1 state is expensive, it is only necessary when running host user space applications, not on each transition between a VM and the hypervisor. For the KVM hypervisor, returning to host user space is already an expensive transition on both ARM and x86. As a result, KVM is heavily optimized to avoid returning to host user space. Measurements presented in Section 5 indicate this overhead is negligible in practice.

Third, to support virtual memory for host user space applications in EL0 and the kernel running in EL2 while preserving normal Linux virtual memory management semantics, we make two Linux modifications. One provides a way to bridge the differences between the different page table formats of EL0 and EL2, and the other uses the single EL2 page table to mimic the behavior using two EL0/EL1 page tables.

Bridging the differences between different page table formats of EL0 and EL2 is important because Linux memory management is designed around the assumption that the same page tables are used from both user and kernel mode, with potentially different access permissions between the two modes. This allows Linux to maintain a consistent per-process view of virtual memory from both the kernel and user space. Violating this assumption would require invasive and complex changes to the Linux kernel. *el2*Linux takes advantage of the fact the differences between EL0/EL1 and EL2 page table formats are relatively small and can be bridged to use the same page tables for both EL0 and EL2 by slightly relaxing a security feature and accepting a higher TLB invalidation frequency on some workloads.

*el2*Linux relaxes a security feature because the EL2 page table format only has a single non-execute bit which must be shared by EL0 and EL2 to use the same page tables for both EL0 and EL2. When setting this bit on a page table entry which is used in both EL2 and EL0, the page is not executable by the kernel or user space, and when clearing this bit, the page is executable by both. Since kernel pages containing code must be executable by the kernel, the single non-execute bit means they end up executable by both user space and the kernel. This problem does not exist for EL1 page tables because they support two bits to control if a page is executable or non-executable, one for EL0 and one for EL1. We emphasize that while this is a slight relaxation of a security feature, it is not a direct security exploit. All kernel pages can still not be read or written from user space, but only executed, and can still only be executed with user privileges. This security relaxation may work against the purpose of kernel hardening techniques such as kernel address space randomization (KASLR), because user software can try to execute random addresses in the kernel's address space and rely on signals to regain control, and by observing the register state of the CPU or by observing other side effects, applications can attempt to reason about where the kernel maps its code and data within its address space.

Alternative solutions exist to support virtual memory for host user space applications in EL0 without relaxing this security feature, but require more invasive changes to Linux. One approach would be to simply not use the same page tables between the kernel and user space and maintain two page tables per process, one used by the host user space in EL0 and one used by the kernel in EL2. This solution would require additional synchronization mechanisms to make sure the two page tables always maintained a consistent view of a process address space between user space threads and the kernel. Another approach would be to not allow Linux to access user space pointers from within the kernel and instead require Linux to translate every user space virtual address into a kernel virtual address by walking the EL0 user space page tables in software from within the kernel on every user access such as read or write system calls that transfer data between user space processes and the kernel.

*el2*Linux may incur a higher TLB invalidation frequency because virtual memory accesses performed in EL2 are not tagged with an Address Space Identifier (ASID), which are used to distinguish different address space resolutions in the TLB to avoid having to invalidate TLB entries when changing address spaces, for example when switching between processes. While the kernel address space is shared for all processes, the kernel also some times accesses user space addresses when copying data between user space, for example when handling system calls. Such accesses should be tagged with the process ASID to ensure that TLB entries only match for the right process. Since memory accesses performed in EL2 are not associated with a ASID, we must invalidate all EL2 entries in the TLB when switching between processes. This does not affect TLB entries for memory accesses done by user space applications, as these still run in EL0 and all EL0 accesses still use ASIDs. We did not observe a slowdown in overall system performance as a result of this design, and estimate that for most virtualization workloads the effect will be minimal, but it could be substantial for other host workloads. Note that VHE hardware uses ASIDs in EL2 and does not have this limitation.

Finally, *el2*Linux uses an approach similar to x86 Linux to enable a single EL2 page table to mimic the behavior using two EL0/EL1 page tables. Instead of having separate page tables for user and kernel address spaces as is done in EL1, *el2*Linux splits a single address space so that half is for user space and the other half is for a shared kernel space among all processes. Similar to x86 Linux, *el2*Linux only maintains a single copy of the second level page tables for the kernel and points to these from the first level page table across all processes. ARM supports a maximum of 48 bits of contiguous virtual addresses, resulting in a maximum of 47 bits of address space for both the kernel and each user space process.

4 Hypervisor Redesign

While running the hypervisor OS kernel in a separate CPU mode is a key aspect of our approach, it turns out that this alone is insufficient to significantly improve virtualization performance, as we will show in Section 5. The hypervisor itself must also be redesigned to take advantage of not having to multiplex the same CPU mode between the hypervisor OS kernel and the VM. We redesigned KVM/ARM based on this insight. A key challenge was to do this in such a way that our modifications could be accepted by the Linux community,

which required also supporting legacy systems in which users may still choose to run the hypervisor OS kernel in EL1. We describe three techniques we used to redesign KVM/ARM's execution flow to improve performance.

First, we redesigned KVM/ARM to avoid saving and restoring EL1 registers on every transition between a VM and the hypervisor. The original KVM/ARM had to save and restore EL1 state on every transition because EL1 was shared between a VM's guest OS kernel and the hypervisor OS kernel. Since the hypervisor OS kernel now runs in EL2 and does not use the EL1 state anymore, it can load the VM's EL1 state into CPU registers when it runs the VM's virtual CPU (VCPU) on the physical CPU for the first time. It does not have to save or modify this state again until it runs another VCPU or has to configure its EL1 runtime to run applications in host user space. This entails not only eliminating copying EL1 state to in-memory hypervisor data structures on each transition between a VM and the hypervisor, but also modifying KVM/ARM to directly access the physical CPU for the running VCPU's EL1 register state since the hypervisor data structures may be out of date. To preserve backwards compatibility to also use KVM/ARM without Linux running in EL2, we keep track of whether a VCPU's EL1 registers are loaded onto the physical CPU or stored in memory and direct accesses to EL1 registers in KVM/ARM to the appropriate location using access functions.

Second, we redesigned KVM/ARM to avoid enabling and disabling virtualization features on every transition between the VM and the hypervisor. The original KVM/ARM had to disable virtualization features when running the hypervisor OS kernel so it could have full access to the underlying hardware, but then enable virtualization features when running a VM so it only had restricted access to virtualized hardware. The configuration of virtualization features such as stage 2 translations, virtual interrupts, and traps on sensitive instructions only apply to software running in EL1 and EL0. Since the hypervisor OS kernel now runs in EL2, it automatically has full access to the underlying hardware and the configuration of virtualization features do not apply to it. Instead, the virtualization features simply remain enabled for running VMs in EL1 and EL0, eliminating frequent writes to the group of special EL2 registers that configures the virtualization features. The only time the virtualization features need to be disabled is for running host user space applications and its supporting EL1 runtime, which happens relatively infrequently.

Third, we redesigned KVM/ARM to avoid the use of shared, intermediate data structures between EL1 and EL2. The original KVM/ARM using split-mode virtualization had to communicate across EL1 and EL2 modes via intermediate data structures mapped in both CPU modes because much of the hypervisor functionality was implemented in the hypervisor OS kernel running in EL1 but needed to have some aspect run in EL2 to program EL2 hardware. The hypervisor ends up processing data twice, once in EL1 which results in writing data to an intermediate data structure, and once in EL2 to process the intermediate data structure and program the hardware. Similarly, duplicative processing also happened when intermediate data structures were used to store EL2 state that needed to be read by the hypervisor OS kernel in EL1 but could only be read by the hypervisor in EL2. This complicates the code and results in many conditional statements. To make matters worse, since EL1 and EL2 run in separate address spaces, accessing the intermediate data structures can result in a TLB miss for both EL1 and EL2. Since the hypervisor OS kernel now runs in EL2 together with the rest of KVM/ARM, there is no longer any need for these intermediate data structures. The previously separate logic to interact with the rest of the hypervisor OS kernel and to program or access the EL2 hardware can be combined into a single optimized step, resulting in improved performance.

A prime example of how eliminating the need for intermediate data structures helped was the virtual interrupt controller (VGIC) implementation, which is responsible for handling virtual interrupts for VMs. VGIC hardware state is only accessible and programmable in EL2, however hypervisor functionality pertaining to virtual interrupts relies on the hypervisor OS kernel, which ran in EL1 with the original KVM/ARM. Since it was not clear when running in EL2 what VGIC state would be needed in EL1, the original KVM/ARM would conservatively copy all of the VGIC state to intermediate data structures so it was accessible in EL1, so that, for example, EL1 could save the state to in-memory data structures if it was going to run another VM. Furthermore, the original KVM/ARM would identify any pending virtual interrupts but then could only write this information to an intermediate data structure, which then needed to be later accessed in EL2 to write them into the VGIC hardware.

Since the hypervisor OS kernel now runs in EL2 together with the rest of KVM/ARM, the redesigned KVM/ARM no longer needs to conservatively copy all VGIC state to intermediate data structures, but can instead have the hypervisor kernel access VGIC state directly whenever needed. Furthermore, since the redesign simplified the execution flow, it became clear that some VGIC registers were never used by KVM and thus never needed to be copied, saved, or restored. It turns out that eliminating extra VGIC register accesses is very beneficial because VGIC register accesses are expensive. Similarly, since the hypervisor OS kernel now runs in EL2, there is no need to check for pending virtual interrupts in both EL1 and EL2. Instead these steps can be combined into a single optimized step that also writes them into the VGIC hardware as needed. As part of this redesign, it became clear that the common case that should be made fast is that there are no pending interrupts so only a single simple check should be required. We further optimized this step by avoiding the need to hold locks in the common case, which was harder to do with the original KVM/ARM code base that

had to synchronize access to intermediate data structures.

To maintain backwards compatibility support for systems not running the hypervisor and its host OS kernel in EL2, while not adding additional runtime overhead from conditionally execution almost all operations in the run loop, we take advantage of the static key infrastructure in Linux. Static keys patch the instruction flow at runtime to avoid conditional branches, and instead replaces no-ops with unconditional branches when a certain feature is enabled. During initialization of KVM/ARM, we activate or deactivate the static branch depending on whether KVM/ARM runs in EL2 or EL1. For example, the run loop uses a static branch to decide if it should call the lowvisor to start switching to a VM in EL2, or if it should simply run the VM if the hypervisor is already running in EL2.

5 Experimental Results

We have successfully merged many of our implementation changes in redesigning KVM/ARM into the mainline Linux kernel, demonstrating the viability of our approach. Getting changes accepted into mainline Linux takes time, and as such, our improvements have been merged into mainline Linux over the course of Linux kernel versions v4.5 through v4.8, with remaining changes scheduled to be applied in upcoming kernel versions.

We evaluate the performance of our new hypervisor design using both microbenchmarks and real application workloads on ARM server hardware. Since no VHE hardware is publicly available yet, we ran workloads on non-VHE ARM hardware using *el2*Linux. We expect that *el2*Linux provides a conservative but similar measure of performance to what we would expect to see with VHE since the critical hypervisor execution paths are almost identical between the two, and VHE does not introduce hardware features that would cause runtime overhead from the hardware. In this sense, these measurements provide the first quantitative evaluation of the benefits of VHE, and provide chip designers with useful experimental data to evaluate whether or not to support VHE in future silicon. We also verified the functionality and correctness of our VHE-based implementation on ARM software models supporting VHE. As a baseline for comparison, we also provide results using KVM on x86 server hardware.

ARM measurements were done using a 64-bit ARMv8 AMD Seattle (Rev.B0) server with 8 Cortex-A57 CPU cores, 16 GB of RAM, a 512 GB SATA3 HDD for storage, and a AMD 10 GbE (AMD XGBE) NIC device. For benchmarks that involve a client interfacing with the ARM server, we ran the clients on an x86 machine with 24 Intel Xeon CPU 2.20 GHz cores and 96 GB RAM. The client and the server were connected using 10 GbE and we made sure the interconnecting switch was not saturated during our measurements. x86 measurements were done using Dell PowerEdge r320 servers, each with a 64-bit Xeon 2.1 GHz E5-2450 with 8 physical CPU cores. Hyper-Threading was disabled on the r320 servers to provide a similar hardware configuration to the ARM servers. Each r320 node had 16 GB of RAM, 4 500 GB 7200 RPM SATA RAID5 HDs for storage, and a Dual-port Mellanox MX354A 10 GbE NIC. For benchmarks that involve a client interfacing with the x86 server, we ran the clients on an identical x86 client. CloudLab [10] infrastructure was used for x86 measurements, which also provides isolated 10 GbE interconnect between the client and server.

To provide comparable measurements, we kept the software environments across all hardware platforms and hypervisors the same as much as possible. KVM/ARM was configured with passthrough networking from the VM to an AMD XGBE NIC device using Linux's VFIO direct device assignment framework. KVM on x86 was configured with passthrough networking from the VM to one of the physical functions of the Mellanox MX354A NIC. Following best practices, we configured KVM virtual block storage with `cache=none`. We configured power management features on both server platforms and ensured both platforms were running at full performance. All hosts and VMs used Ubuntu 14.04 with identical software configurations. The client machine used for workloads involving a client and server used the same configuration as the host and VM, but using Ubuntu's default v3.19.0-25 Linux kernel.

We ran benchmarks on bare-metal machines 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 two configurations: (1) running natively on Linux capped at 4 cores and 12 GB RAM, (2) running in a VM using KVM with 8 physical cores and 16 GB RAM with the VM capped at 4 virtual CPUs (VCPUs) and 12 GB RAM. For network related benchmarks, the clients were run natively on Linux and configured to use the full hardware available.

To minimize measurement variability, we pinned each VCPU of the VM to a specific physical CPU (PCPU) and ensured that no other work was scheduled on that PCPU. We also statically allocated interrupts to a specific CPU, and for application workloads in VMs, the physical interrupts on the host system were assigned to a separate set of PCPUs from those running the VCPUs.

We compare across Linux v4.5 and v4.8 on ARM to quantify the impact of our improvements, as the former does not contain any of them while the latter contains a subset of our changes merged into mainline Linux. To ensure that our results are not affected by other changes to Linux between the two versions, we ran both v4.5 and v4.8 Linux natively on both the ARM and x86 systems and compared the results and we found that there were no noticeable differences between these versions of Linux. For comparison purposes, we measured four different system configurations, ARM, ARM EL2, ARM EL2 OPT, and x86. ARM uses vanilla KVM/ARM in Linux v4.5, the kernel version before any of our imple-

Name	Description
Hypercall	Transition from the VM to the hypervisor and return to the VM without doing any work in the hypervisor. Measures bidirectional base transition cost of hypervisor operations.
I/O Kernel	Trap from the VM to the emulated interrupt controller in the hypervisor OS kernel, and then return to the VM. Measures a frequent operation for many device drivers and baseline for accessing I/O devices supported by the hypervisor OS kernel.
I/O User	Trap from the VM to the emulated UART in QEMU and then return to the VM. Measures base cost of operations that access I/O devices emulated in the hypervisor OS user space.
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.

Table 1: Microbenchmarks

mentation changes were merged into Linux. ARM EL2 uses the same KVM/ARM in Linux v4.5 but with modifications to run *el2*Linux to quantify the benefits of running Linux in EL2 without also redesigning the KVM/ARM hypervisor itself. ARM EL2 OPT uses our redesigned KVM/ARM in Linux v4.8, including all of the optimizations described in this paper, both those already merged into Linux v4.8 and those scheduled to be applied in upcoming Linux versions.

5.1 Microbenchmark Results

We first ran various microbenchmarks as listed in Table 1, which are part of the KVM unit test framework [23]. We slightly modified the test framework to measure the cost of virtual IPIs and to obtain cycle counts on the ARM platform to ensure detailed results by configuring the VM with direct access to the cycle counter. Table 2 shows the microbenchmark results. Measurements are shown in cycles instead of time to provide a useful comparison across server hardware with different CPU frequencies.

The Hypercall measurement quantifies the base cost of any operation where the hypervisor must service the VM. Since KVM handles hypercalls in the host OS kernel, this metric also represents the cost of transitioning between the VM and the hypervisor OS kernel. For Hypercall, the ARM EL2 OPT is a mere 12% of the ARM cost and roughly 50% of the x86 cost, measured in cycles. Comparing the ARM and ARM EL2 costs, we see that only running the hypervisor OS kernel in a separate CPU mode from the VM kernel does not by itself yield much improvement. Instead, redesigning the hypervisor to take advantage of this fact is essential to obtain a significant performace improvement as shown by the ARM EL2 OPT costs.

The I/O Kernel measurement quantifies the cost of I/O requests to devices supported by the hypervisor OS kernel. The cost consists of the base Hypercall cost plus doing some work in the hypervisor OS kernel. For I/O Kernel, the ARM EL2 OPT cost is only 20% of the original ARM cost because of the significant improvement in the Hypercall cost component of the overall I/O Kernel operation.

Microbenchmark	ARM	ARM EL2	ARM EL2 OPT	x86
Hypercall	6,413	6,277	752	1,437
I/O Kernel	8,034	7,908	1,604	2,565
I/O User	10,012	10,186	7,630	6,732
Virtual IPI	13,121	12,562	2,526	3,102

Table 2: Microbenchmark Measurements (cycle counts)

The I/O User measurement quantifies the cost of I/O requests that are handled by host user space. For I/O User, ARM EL2 OPT cost is only reduced to 76% of the ARM cost. The improvement is less in this case because our *el2*Linux implementation requires restoring the host's EL1 state before returning to user space since running user applications in EL0 without VHE uses an EL1 runtime, as discussed in Section 3.2. However, returning to user space from executing the VM has always been known to be slow, as can also be seen with the x86 I/O User measurement in Table 2. Therefore, most hypervisor configurations do this very rarely. For example, the vhost configuration of virtio [25] paravirtualized I/O that is commonly used with KVM completely avoids going to host user space when doing I/O.

Finally, the Virtual IPI measurement quantifies the cost of issuing virtual IPIs (Inter Processor Interrupts), a frequent operation in multi-core OSes. It involves exits from both the sending VCPU and receiving VCPU. The sending VCPU exits because sending an IPI traps and is emulated by the underlying hypervisor. The receiving VCPU exits because it gets a physical interrupt which is handled by the hypervisor. For Virtual IPI, ARM EL2 OPT cost is only 19% of the original ARM cost because of the significant improvement in the Hypercall cost, which benefits both the sending and receiving VCPUs in terms of lower exit costs.

Our microbenchmark measurements show that our KVM/ARM redesign is roughly an order of magnitude faster than KVM/ARM's legacy split-mode design in transitioning between the VM and the hypervisor. The ARM EL2 numbers show slight improvement over the ARM numbers, due to the removal of the double trap cost [14] introduced by split-mode virtualization. However, a key insight based on our implementation experience and these results is that only running the hypervisor OS kernel in a separate CPU mode from the VM kernel is insufficient to have much of a performance benefit, even on architectures like ARM which have the ability to quickly switch between the two separate CPU modes without having to multiplex any state. However, if the hypervisor is designed to take advantage of running the hypervisor OS kernel in a separate mode, and the hardware provides the capabilities to do so and to switch quickly between the two modes, then the cost of low-level VM-to-hypervisor interactions can be much lower than on

Name	Description
Kernbench	Compilation of the Linux 3.17.0 kernel using the allnoconfig for ARM using GCC 4.8.2.
Hackbench	`hackbench` [24] using Unix domain sockets and 100 process groups running with 500 loops.
Netperf	`netperf` v2.6.0 [22] starting netserver on the server and running with its default parameters on the client in three modes: TCP_STREAM, TCP_MAERTS, and TCP_RR, measuring throughput and latency, respectively.
Apache	`Apache` v2.4.7 Web server with a remote client running `ApacheBench` [28] v2.3, 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.

Table 3: Application Benchmarks

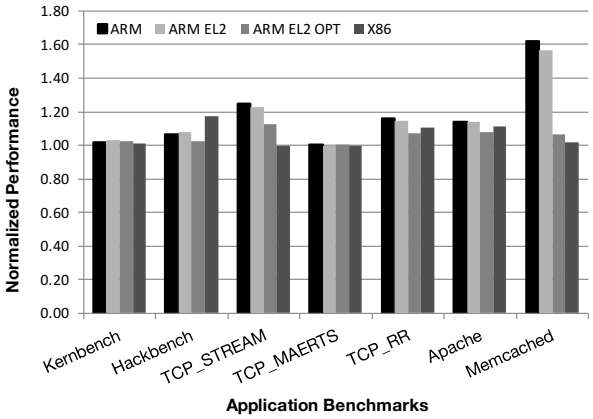

Figure 2: Application Benchmark Performance

systems like x86, even though they have highly optimized VM Entry and Exit hardware mechanisms to multiplex a single CPU mode between the hypervisor and the VM.

5.2 Application Benchmark Results

We next ran a mix of widely-used CPU and I/O intensive application workloads as listed in Table 3. 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. Figure 2 shows the relative performance overhead of executing in a VM compared to natively without virtualization.

We normalize measurements to native execution for the respective platform, with one being the same as native performance. ARM numbers are normalized to native execution on the ARM platform, and x86 numbers are normalized to native execution on the x86 platform. Lower numbers mean less overhead and therefore better overall performance. We focus on normalized overhead as opposed to absolute performance since our goal is to improve VM performance by reducing the overhead from intervention of the hypervisor and from switching between the VM and the hypervisor OS kernel.

Like the microbenchmark measurements in Section 5.1, the application workload measurements show that ARM EL2 performs similarly to ARM across all workloads, showing that running the hypervisor OS kernel in a separate CPU mode from the VM kernel without changing the hypervisor does not benefit performance much. The ARM EL2 OPT results, however, show significant improvements across a wide range of applications workloads.

For cases in which original ARM did not have much overhead, ARM EL2 OPT performs similarly to original ARM as there was little room for improvement. For example, Kernbench runs mostly in user mode in the VM and seldom traps to the hypervisor, resulting in very low overhead on both ARM and x86. However, the greater the initial overhead for original ARM, the greater the performance improvement achieved with ARM EL2 OPT. For example, original ARM incurs more than 60% overhead for Memcached while ARM EL2 OPT reduces that overhead by more than five times to roughly 10% compared to native execution. Memcached causes frequent traps to the hypervisor OS kernel to process, configure, and forward physical interrupts. As a result, this workload benefits greatly from the much reduced hypercall cost for ARM EL2 OPT compared to original ARM. As another example, original ARM incurs roughly 15% overhead for Apache while ARM EL2 OPT reduces that overhead by roughly 50% to 8% compared to native execution, which is even smaller than x86. Apache requires processing network interrupts and sending virtual IPIs, both of which benefit from the reduced hypercall cost for ARM EL2 OPT.

It is instructive to take a closer look at the various Netperf measurements, TCP_STREAM, TCP_RR and TCP_MAERTS, which show ARM EL2 OPT providing different performance improvements over original ARM and x86. Since we use passthrough to directly assign the network device to the VM, the primary source of overhead comes from interrupt handling because the VM can otherwise directly program the device without intervention from the hypervisor. The network devices used on both the ARM and x86 servers generate physical RX interrupts when receiving network data, which is the primary operation of TCP_STREAM and TCP_RR. These physical interrupts are handled by VFIO in the host kernel and KVM must forward them as virtual interrupts to the VM, which results in execution overhead. The driver for the AMD XGBE NIC used in the ARM server frequently masks and unmasks interrupts for this device due to driver implementation details and support for NAPI, which switches between interrupt driven and polling mode for the VM network driver. On the other hand, the driver for the Mellanox NIC used in the x86 server does not enable and disable IRQs using the interrupt controller, but instead manages masking of interrupts at the device level, which avoids traps to the hypervisor for these operations because the device is directly assigned to the VM.

TCP_STREAM is a throughput benchmark and since

x86 has fewer traps to the hypervisor than ARM due to these NIC diferences, x86 has lower virtualization overhead than any ARM configuration, including ARM EL2 OPT. The same explanation applies to Memcached as well. The TCP_RR workload is a latency measurement benchmark, which sends a single network packet back and forward between the client and the server in serial, and every single packet causes an interrupt for both ARM and x86, resulting in overhead on both platforms. Since ARM EL2 OPT has lower transition costs between the VM and hypervisor when comparing against either original ARM or x86, it also ends up having the lowest overhead for TCP_RR. For both TCP_STREAM and TCP_RR, ARM EL2 OPT reduces the overhead of original ARM by approximately 50% as a result of the reduced cost of transitioning between the hypervisor OS kernel and the VM when masking and unmasking virtual interrupts, and when forwarding physical interrupts as virtual interrupts, respectively. TCP_MAERTS shows almost no overhead for all configurations, because sending packets from the VM to the client generates almost no interrupts and the VMs can access the devices directly because of their passthrough device configuration.

6 Related Work

Virtualization on x86 started with software-only approaches [3, 7], but as virtualization became increasingly important, Intel introduced VMX hardware virtualization support to run VMs with unmodified guest OSes and eliminate the need for binary translation and CPU paravirtualization [29]. Initially, hypervisors using hardware virtualization support did not provide good performance [1], but as the hardware support matured and provided additional features like EPT, performance improved using VMX. Much other work has been also done on analyzing and improving the performance of x86 virtualization [27, 26, 2, 18, 9, 16, 19, 6], but none of these techniques addressed the core issue of the cost of sharing kernel mode across guest and hypervisor OS kernels.

Full-system virtualization of the ARM architecture in some ways mirrors the evolution of x86 virtualization. Early approaches were software only, could not run unmodified guest OSes, and often suffered from poor performance [20, 13, 15, 4]. As virtualization became increasingly important on ARM, ARM introduced hardware virtualization support to run VMs with unmodified guest OSes, but ARM took a very different approach to CPU virtualization that made it difficult to implement popular hypervisors such as KVM due to mismatches between the hardware support and assumptions about the software design [14]. As a result, ARM hypervisor implementations have much higher costs for many common VM operations than their x86 counterparts [12]. We show that by taking advantage of the additional CPU mode provided by ARM VE to run not only the hypervisor but also its OS kernel, in conjunction with a redesign of the hypervisor itself, it is possible to achieve superior VM performance on ARM versus x86.

7 Conclusions

Although ARM and x86 architectural support for virtualization are quite different, previous hypervisors across both architectures shared a common limitation; the need to share kernel mode state between the host OS kernel used by the hypervisor and guest OS kernels used in VMs. Our work shows for the first time how, with the proper architectural support and hypervisor design, the hypervisor and its OS kernel can be run in a separate CPU mode from VMs, avoiding the cost of multiplexing shared CPU state between the hypervisor and VMs. We show how this codesign of hardware and software support for virtualization can be used to reimplement an existing hypervisor, KVM/ARM, with evolutionary changes to the code base without requiring a clean slate implementation. This approach was essential in allowing us to merge many of our changes into mainline Linux. We show that our approach can be implemented using currently available ARM hardware, and that new hardware features in future ARM architecture versions can be used to support this approach without any changes to Linux other than to KVM/ARM itself. We show that our KVM/ARM redesign can provide an order of magnitude performance improvement over previous versions of KVM/ARM on key hypervisor operations. We also show that the combination of hardware and software virtualization support on ARM can provide roughly two times better performance than its counterpart on x86. Our results indicate that running the hypervisor and its hypervisor OS kernel in a separate CPU mode from the VMs as possible on ARM can provide superior performance to x86 approaches because it allows for faster transitions between the hypervisor and VMs. As virtualization continues to be of importance, our work provides an important counterpoint to x86 practices which we believe is instrumental in designing future virtualization support for new architectures.

8 Acknowledgments

Marc Zyngier implemented some VGIC optimizations and supported our efforts to upstream our improvements to KVM/ARM. Ard Biesheuvel helped us understand the virtual memory changes needed to run Linux in EL2. Eric Auger implemented VFIO passthrough support for ARM and provided help in configuring passthrough on ARM server hardware. Paolo Bonzini and Alex Williamson helped analyze KVM x86 performance. Mike Hibler provided support for system configurations in CloudLab. This work was supported in part by Huawei Technologies and NSF grants CNS-1422909, CNS-1563555, and CCF-1162021.

References

[1] ADAMS, K., AND AGESEN, O. 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.

[2] AGESEN, O., MATTSON, J., RUGINA, R., AND SHELDON, J. Software Techniques for Avoiding Hardware Virtualization Exits. In *Proceedings of the 2012 USENIX Annual Technical Conference* (June 2012), pp. 373–385.

[3] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Oct. 2003), pp. 164–177.

[4] BARR, K., BUNGALE, P., DEASY, S., GYURIS, V., HUNG, P., NEWELL, C., TUCH, H., AND ZOPPIS, B. The VMware Mobile Virtualization Platform: is that a hypervisor in your pocket? *SIGOPS Operating Systems Review 44*, 4 (Dec. 2010), 124–135.

[5] BONZINI, P. Virtualizing the locomotive, Sept. 2005. `https://lwn.net/Articles/657282/`.

[6] BUELL, J., HECHT, D., HEO, J., SALADI, K., AND TAHERI, H. R. Methodology for Performance Analysis of VMware vSphere under Tier-1 Applications. *VMware Technical Journal 2*, 1 (June 2013), 19–28.

[7] BUGNION, E., DEVINE, S., ROSENBLUM, M., SUGERMAN, J., AND WANG, E. Y. Bringing Virtualization to the x86 Architecture with the Original VMware Workstation. *ACM Transactions on Computer Systems 30*, 4 (Nov. 2012), 12:1–12:51.

[8] BUGNION, E., NIEH, J., AND TSAFRIR, D. *Hardware and Software Support for Virtualization*, vol. 12 of *Synthesis Lectures on Computer Architecture*. Morgan & Claypool Publishers, Feb. 2017.

[9] CHERKASOVA, L., AND GARDNER, R. Measuring CPU Overhead for I/O Processing in the Xen Virtual Machine Monitor. In *Proceedings of the 2005 USENIX Annual Technical Conference* (May 2005), pp. 387–390.

[10] CLOUDLAB. `http://www.cloudlab.us`.

[11] DALL, C., AND LI, S.-W. *el2*linux. `https://github.com/chazy/el2linux`.

[12] DALL, C., LI, S.-W., LIM, J. T., NIEH, J., AND KOLOVENTZOS, G. ARM Virtualization: Performance and Architectural Implications. In *Proceedings of the 43rd International Symposium on Computer Architecture* (2016), pp. 304–316.

[13] DALL, C., AND NIEH, J. KVM for ARM. In *Proceedings of the Ottawa Linux Symposium* (July 2010), pp. 45–56.

[14] DALL, C., AND NIEH, J. 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.

[15] DING, J.-H., LIN, C.-J., CHANG, P.-H., TSANG, C.-H., HSU, W.-C., AND CHUNG, Y.-C. ARMvisor: System Virtualization for ARM. In *Proceedings of the Ottawa Linux Symposium* (July 2012), pp. 93–107.

[16] GAMAGE, K., AND KOMPELLA, X. Opportunistic Flooding to Improve TCP Transmit Performance in Virtualized Clouds. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (Oct. 2011), pp. 24:1–24:14.

[17] GOLDBERG, R. P. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, Cambridge, MA, 1972.

[18] GORDON, A., AMIT, N., HAR'EL, N., BEN-YEHUDA, M., LANDAU, A., SCHUSTER, A., AND TSAFRIR, D. ELI: Bare-Metal Performance for I/O Virtualization. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems* (Feb. 2012), pp. 411–422.

[19] HEO, J., AND TAHERI, R. Virtualizing Latency-Sensitive Applications: Where Does the Overhead Come From? *VMware Technical Journal 2*, 2 (Dec. 2013), 21–30.

[20] HWANG, J., SUH, S., HEO, S., PARK, C., RYU, J., PARK, S., AND KIM, C. 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.

[21] INTEL CORPORATION. Intel 64 and IA-32 Architectures Software Developer's Manual, 325384-056US, Sept. 2015.

[22] JONES, R. Netperf. `http://www.netperf.org/netperf/`.

[23] KIVITY, A. KVM Unit Tests. http://www.linux-kvm.org/page/KVM-unit-tests.

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

[25] RUSSELL, R. virtio: Towards a De-Facto Standard for Virtual I/O Devices. *SIGOPS Operating Systems Review 42*, 5 (July 2008), 95–103.

[26] SANTOS, J. R., TURNER, Y., JANAKIRAMAN, G. J., AND PRATT, I. Bridging the Gap between Software and Hardware Techniques for I/O Virtualization. In *Proceedings of the 2008 USENIX Annual Technical Conference* (June 2008), pp. 29–42.

[27] SUGERMAN, J., VENKITACHALAM, G., AND LIM, B.-H. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference* (June 2001), pp. 1–14.

[28] THE APACHE SOFTWARE FOUNDATION. ab. http://httpd.apache.org/docs/2.4/programs/ab.html.

[29] UHLIG, R., NEIGER, G., RODGERS, D., SANTONI, A. L., MARTINS, F. C. M., ANDERSON, A. V., BENNETT, S. M., KAGI, A., LEUNG, F. H., AND SMITH, L. Intel Virtualization Technology. *IEEE Computer 38*, 5 (May 2005), 48–56.

[30] XEN ARM WITH VIRTUALIZATION EXTENSIONS. http://wiki.xenproject.org/wiki/Xen_ARM_with_Virtualization_Extensions.