The Design, Implementation, and Evaluation of Software and Architectural Support for ARM Virtualization

Christoffer Dall

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

COLUMBIA UNIVERSITY

2018

ABSTRACT

The Design, Implementation, and Evaluation of Software and Architectural Support for ARM Virtualization

Christoffer Dall

The ARM architecture is dominating in the mobile and embedded markets and is making an upwards push into the server and networking markets where virtualization is a key technology. Similar to x86, ARM has added hardware support for virtualization, but there are important differences between the ARM and x86 architectural designs. Given two widely deployed computer architectures with different approaches to hardware virtualization support, we can evaluate, in practice, benefits and drawbacks of different approaches to architectural support for virtualization.

This dissertation explores new approaches to combining software and architectural support for virtualization with a focus on the ARM architecture and shows that it is possible to provide virtualization services an order of magnitude more efficiently than traditional implementations.

First, we investigate why the ARM architecture does not meet the classical requirements for virtualizable architectures and present an early prototype of KVM for ARM, a hypervisor using lightweight paravirtualization to run VMs on ARM systems without hardware virtualization support. Lightweight paravirtualization is a fully automated approach which replaces sensitive instructions with privileged instructions and requires no understanding of the guest OS code.

Second, we introduce split-mode virtualization to support hosted hypervisor designs using ARM's architectural support for virtualization. Different from x86, the ARM virtualization extensions are based on a new hypervisor CPU mode, separate from existing CPU modes. This separate hypervisor CPU mode does not support running existing unmodified OSes, and therefore hosted hypervisor designs, in which the hypervisor runs as part of a host OS, do not work on ARM. Split-mode virtualization splits the execution of the hypervisor such that the host OS with core hypervisor functionality runs in the existing kernel CPU mode, but a small runtime runs in the hypervisor CPU mode and supports switching between the VM and the host OS. Split-mode virtualization was used in KVM/ARM, which was designed from the ground up as an open source project and merged in the mainline Linux kernel, resulting in interesting lessons about translating research ideas into practice.

Third, we present an in-depth performance study of 64-bit ARMv8 virtualization using server hardware and compare against x86. We measure the performance of both standalone and hosted hypervisors on both ARM and x86 and compare their results. We find that ARM hardware support for virtualization can enable faster transitions between the VM and the hypervisor for standalone hypervisors compared to x86, but results in high switching overheads for hosted hypervisors compared to both x86 and to standalone hypervisors on ARM. We identify a key reason for high switching overhead for hosted hypervisors being the need to save and restore kernel mode state between the host OS kernel and the VM kernel. However, standalone hypervisors such as Xen, cannot leverage their performance benefit in practice for real application workloads. Other factors related to hypervisor software design and I/O emulation play a larger role in overall hypervisor performance than low-level interactions between the hypervisor and the hardware.

Fourth, realizing that modern hypervisors rely on running a full OS kernel, the hypervisor OS kernel, to support their hypervisor functionality, we present a new hypervisor design which runs the hypervisor and its hypervisor OS kernel in ARM's separate hypervisor CPU mode and avoids the need to multiplex kernel mode CPU state between the VM and the hypervisor. Our design benefits from new architectural features, the virtualization host extensions (VHE), in ARMv8.1 to avoid modifying the hypervisor OS kernel to run in the hypervisor CPU mode. We show that the hypervisor must be co-designed with the hardware features to take advantage of running in a separate CPU mode and implement our changes to KVM/ARM. We show that running the hypervisor OS kernel in a separate CPU mode from the VM and taking advantage of ARM's ability to quickly switch between the VM and hypervisor results in an order of magnitude reduction in overhead for important virtualization microbenchmarks and reduces the overhead of real application workloads by more than 50%.

Contents

List of Figures

List of Tables

Acknowledgements

First I owe huge thanks to my advisor, Jason Nieh. Without his initial belief in me, I would not have had the opportunity to work with something I love, at a level I only dreamed was possible. I owe much of the best writing in the research papers on which much of this dissertation is based to him. Jason's unparalleled attention to details and tireless effort to bring our work to the highest quality is inspiring and has given me something to strive for in my own work for the rest of my professional life. In particular, I envy his energy to ensure that no question remains unanswered. Jason has spent countless hours on high-bandwidth discussions about most aspects of my research, discussing even the most subtle points in great depth, which gave me insights into my own work that I would have never received without his help. Working with Jason has been a most challenging and rewarding experience, and I will joyfully remember the wholehearted laughs we shared during many research discussions.

My time at Columbia gave me the opportunity to work with some fantastic minds and individuals. Jeremy Andrus showed me the meaning of working hard, and showed me that plowing through endless levels of complexity and diving into seemingly impossible tasks head first, can lead to great success. Even in the most stressful situations, Jeremy remained calm and would always express a clear path forward. Jeremy was always a friend in times of need, always willing to lend a helping hand, and never stopped to impress me with his ability to be devoted to both family and professional life. Nicolas Viennot possesses one of the brightest minds I have ever had the pleasure of working with. His ability to process multi-faceted problems at multiple levels while always preserving a pragmatic perspective is outstanding, and he helped me more than once in solving highly complex challenges. Jintack Lim and Shih-Wei Li joined me in my work on ARM virtualization during their first years as students in the Columbia computer science department and helped me carry out experiments, analyze results, and debug problems. Without their help and hard work, many aspects of my work would have never materialized. Several professors at Columbia gave me valuable feedback on

versions of research papers and dedicated time to serve on my dissertation committee, for which I am very thankful.

Much of my work was done in interaction with the Linux and QEMU open-source communities. I owe huge thanks to Marc Zyngier who implemented large parts of KVM/ARM, including the 64-bit support. Marc and I now co-maintain KVM/ARM in the Linux kernel together in what can only be described as an extraordinarily pleasant, inspiring, and productive collaboration, which has led to a warm friendship as well. Marc was always helpful in bridging research, open-source, and industry perspectives. Peter Maydell helped me understand QEMU, and lent his expertise in modeling systems and rigorously understanding the ARM architecture to me and the wider communities on numerous occasions. Rusty Russell worked on the original coprocessor user space interface for KVM/ARM and assisted with upstreaming. Will Deacon and Avi Kivity provided numerous helpful code reviews of the initial KVM/ARM submission. Catalin Marinas helped establish initial contact with the ARM kernel development team and provided input on memory virtualization work. Paolo Bonzini helped me better understand KVM x86 performance. Alexander Graf helped with initial questions in developing the first KVM/ARM prototype. Ian Campbell and Stefano Stabellini helped me understand Xen internals and advised on how to develop a measurement framework for Xen ARM.

Finally, I wish to thank friends and family who helped me realize this work. My wife Lea, who always gives me unconditional support, makes me capable of more than I imagined. My son Halfdan inspires me to do even more. Uffe Black Nielsen encouraged me to keep going in dark moments and listened patiently to my challenges over the years. Many other friends and colleagues provided occasional support and comfort, and I remain grateful to all of those who helped me in this process.

Preface

This document is submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Graduate School of Arts and Sciences.

Throughout this dissertation, the first person “I”, is the default voice and is used whenever the text speaks about my own conclusions and explanations, because I take full ownership and responsibility for that content and for any inaccuracies expressed in this dissertation. I use “we” when referring to expected conclusions occurring between me as the author and the reader, such as when explaining results or arriving at logical conclusions. In this case, “we” refers to “me and the reader”. I also use ”we”, whenever I speak about work or results produced in collaboration with others, for example results from papers and articles co-authored with peers and colleagues.

Introduction

A key technology across a wide range of computing is the ability to run Virtual Machines (VMs). Virtualization decouples the hardware from the operating system (OS) by inserting an additional layer of software, the hypervisor, between the hardware and the OS. The hypervisor multiplexes the hardware and creates several VM abstractions with interfaces similar to the real machine. The same OS software, originally written to run directly on the hardware, can then run inside VMs. Virtualization has many benefits. Physical hardware resources can be much better utilized by multiplexing them between several isolated VMs, VMs can be seamlessly migrated to avoid downtime when servicing hardware, and users can share cloud infrastructure without having to trust each other. Virtualization underpins modern cloud computing, and is ubiquitous in enterprise servers. Obviously, constructing efficient and reliable hypervisors on today's computing platforms is of key importance.

The classically used technique to construct hypervisors, trap-and-emulate, works by running the entire VM, including the *guest OS* running inside the VM, in the CPU's unprivileged user mode, and running the hypervisor in the CPU's privileged kernel mode. Whenever the VM attempts to operate on the physical machine, the hardware traps from the VM to the hypervisor, and the hypervisor emulates the intended operation in software; trap-and-emulate. However, this only works if the hardware does in fact trap all relevant operations from the VM running in user mode to the hypervisor running in kernel mode. This is unfortunately not always the case, because the CPU's protection mechanism is designed to protect the OS from user applications and not to support virtualization. Some computer architectures, virtualizable architectures [66], trap all relevant operations when executed in user mode to kernel mode, but others, non-virtualizable architectures, do not. For example, some operations are simply ignored when executed in user mode instead of trapping to kernel mode.

The dominant architecture across desktops, laptops, and servers, x86, did not satisfy the trap-and-emulate criteria, which was a key challenge in virtualizing x86 CPUs. Instead, when x86 CPUs

became powerful enough to run multiple VMs, virtualization vendors took to more demanding approaches. Instead of directly executing the unmodified guest OS in the VM, VMware used binary translation to combine interpretation with direct execution [24] and successfully ran VMs on x86 with reasonable performance. As virtualization grew even more in popularity, paravirtualization [83, 82] saw the light of day. Paravirtualization exposes an abstract machine interface to the VM, different from the real machine, and modifies the guest OS to operate on this abstract interface, sacrificing support for closed-source OSes in favor of better performance [16]. Both techniques are unattractive for a variety of reasons. Binary translation is incredibly difficult to implement efficiently, and paravirtualization solutions are difficult to maintain, because modified, or *paravirtualized*, copies of all supported versions of guest OSes must be maintained. Both techniques are also significantly more complicated in their implementations compared to pure trap-and-emulate. Recognizing these challenges as well as the growing and persistent need for virtualization, Intel eventually introduced hardware virtualization support to x86 [80].

Modern x86 hypervisors most commonly rely on hardware support for virtualization. In fact, the Linux Kernel-based Virtual Machine (KVM), is a hypervisor initially developed exclusively for the x86 virtualization features. Other hypervisors originally developed to use binary translation or paravirtualization have also moved towards relying on hardware virtualization support. Much work has been done in studying the benefits of various types of hypervisor designs and analyzing and improving the performance of virtualization. However, almost all recent work was done using x86 and makes implicit assumptions about hardware virtualization based on a single architecture.

The ARM architecture has long ago established itself as the architecture of choice across mobile and embedded systems, leveraging its benefits in customizability and power efficient implementations in these markets. But ARM CPUs also continue to increase their performance such that they are now within the range of x86 CPUs for many classes of applications. This is spurring the development of new ARM-based servers and an upward push of ARM CPUs into other areas such as networking systems and high-performance computing (HPC). Therefore, similar to x86, recent ARM CPUs now include hardware support for virtualization, but there are important differences between the ARM and x86 architectural support. These differences impact both hypervisor design and performance, and many conclusions from x86-based virtualization research must be revisited in the light of an alternative architectural approach to supporting virtualization.

Given the complexity of the x86 CISC-style architecture and protection mechanism, Intel designed their virtualization support by duplicating the entire CPU state, including all the privilege levels of the CPU, into two separate *operations*, one for the hypervisor and one for VMs, and provided hardware mechanisms to atomically switch between these two operations. ARM, on the other hand, is known for its relatively clean RISC-style architecture and its clearly defined CPU privilege hierarchy. Instead of duplicating the entire CPU state like x86, ARM added a new CPU mode with its own privilege level designed specifically to run hypervisors. These two different approaches encourage very different hypervisor software designs. Where the x86 approach of leaving the existing protection mechanisms intact is designed with maximum compatibility of existing software stacks in mind, ARM clearly imagined small standalone hypervisors running in their new hypervisor CPU mode.

We have now, for the first time, two widely deployed computer architectures which both include support for virtualization, and where their approaches to architectural support differ at their very core. This allows us to evaluate, in practice, benefits and drawbacks of each approach, and we can extract higher level conclusions about designing both hardware virtualization features and designing hypervisors that use them. The x86 approach of duplicating the entire CPU state is well suited for multiple hypervisor designs, but performance is restricted by the cost of the hardware mechanism used to switch between the hypervisor and VM operations. ARM, on the other hand, provides a separate CPU mode with a separate set of registers and can switch very quickly between two CPU modes. However, ARM initially focused exclusively on supporting simple standalone hypervisors. In practice though, standalone, or Type 1, hypervisors can be difficult to deploy in the ARM ecosystem. The ARM ecosystem is much more diverse than x86, has no legacy PC standard, and many ARM implementations, especially for the embedded and mobile markets, do not follow any standards beyond the instruction set architecture (ISA). Hosted, or Type 2, hypervisors, which integrate with an existing host OS, are easier to deploy, because they benefit from existing hardware support in the host OS, and don't need to be ported to every supported platform.

While small standalone hypervisors can potentially reduce the size of the trusted computing base (TCB), hosted hypervisors are easier to deploy and can share the maintenance cost with the host OS. As a result, both types of hypervisors are useful, depending on the deployment scenario for virtualization. For example, small standalone hypervisors may be more suitable for safety crit-

ical deployments, and hosted hypervisors may be more suitable for general purpose computing deployments such as servers with virtualization support. Hardware support for virtualization should therefore support both types of hypervisors, and provide as low overhead as possible regardless of the hypervisor software design.

This dissertation explores new approaches to combining software and architectural support for virtualization with a focus on the ARM architecture and shows that it is possible to provide virtualization services an order of magnitude more efficiently than traditional implementations.

First, I survey the ARM architecture before virtualization support was added and identify several violations of the classical requirements for virtualizable architectures. I then present an early prototype of KVM for ARM, which uses *lightweight paravirtualization*. Lightweight paravirtualization is a script-based method to automatically modify the source code of the guest operating system kernel to issue calls to the hypervisor when needed. It differs from traditional paravirtualization in only minimally modifying the guest OS, being fully automated, and requiring no knowledge or understanding of the guest operating system kernel code. Lightweight paravirtualization is architecture specific, but operating system independent. In contrast, traditional paravirtualization, which is both architecture and operating system dependent, requires detailed understanding of the guest operating system kernel to know how to modify its source code, and then requires ongoing maintenance and development to maintain heavily modified versions of operating systems that can be run in VMs.

Second, I present my experiences building KVM/ARM, the first hypervisor that leverages ARM hardware virtualization support to run unmodified guest OSes on ARM multicore hardware. With KVM/ARM, we introduce split-mode virtualization, a new approach to hypervisor design that splits the core hypervisor so that it runs across different privileged CPU modes to take advantage of the specific benefits and functionality offered by each CPU mode. This approach addresses a limitation in the initial ARM architectural support for virtualization, which was designed for standalone hypervisors and does not easily support hosted hypervisor designs. Hosted hypervisors, which leverage existing hardware support from their host OS, have important benefits over standalone hypervisors in the very diverse ARM ecosystem, which lacks legacy and standards compared to x86. Split-mode virtualization allows hosted hypervisors such as KVM, which integrates with the Linux kernel and reuses its hardware support, to be used on ARM. KVM/ARM is the ARM hypervisor in the mainline Linux kernel and was built from the ground up as an open source project, specifically targeting

acceptance into mainline Linux, and was merged in the Linux kernel version 3.9, making it the de facto standard Linux ARM hypervisor available for real world use.

Third, I present an in-depth study of ARM virtualization performance on multi-core server hardware. We measured the performance of the two most popular ARM hypervisors, KVM/ARM using split-mode virtualization as described above and Xen, and compared them with their respective x86 counterparts. KVM, a hosted hypervisor, and Xen, a standalone hypervisors, both support both ARM and x86, but because of the substantial differences between the ARM and x86 architectural support for virtualization, their designs vary greatly across the two architectures. Prior to our work, it was unclear whether these differences had a material impact, positive or negative, on performance. Our performance study helps hardware and software architects building efficient ARM virtualization solutions, and allows companies to evaluate how best to deploy ARM virtualization solutions to meet their infrastructure needs.

Fourth, based on the results of the performance study of ARM virtualization performance which revealed significantly higher overhead for specific hypervisor operations on hosted compared to standalone hypervisors on ARM, we present a new hypervisor design to improve the performance of hosted hypervisors on ARM. Our new hypervisor design takes advantage of unique features of the ARM architecture and can significantly improve the performance of hosted hypervisors based on a key insight; modern hypervisors rely on running full OS kernels to support their hypervisor functionality and manage underlying hardware. Our new design runs both the hypervisor and its hypervisor OS kernel in a CPU mode separate from the normal kernel mode used to run VM kernels. That way, kernel mode is no longer shared between the hypervisor OS kernel and the VM kernel, and kernel mode register state does not have to be multiplexed by saving and restoring it on each transition between the VM and the hypervisor. Instead, the hardware can trap directly from the VM to the hypervisor OS kernel without having to save and restore any state, resulting in improved overall performance. This new hypervisor design takes advantage of recently added features to the ARM architecture, the Virtualization Host Extensions (VHE) introduced in ARMv8.1 [13]. We show that by redesigning the hypervisor to take advantage of running full OSes in separate CPU modes and achieving very fast switching between the VM and the hypervisor, we can show more than a 50% reduction in virtualization overhead in real application workloads, and an order of magnitude reduction in important microbenchmarks.

Contributions

This dissertations represents significant engineering efforts, novel research ideas, and important results relating to architectural support for virtualization and ARM virtualization in particular. Prior to this work, ARM virtualization was an almost unexplored area and there were no publicly available software solutions. Today, in part due the work presented here, ARM virtualization is a well-understood area with widespread adoption and an active industry. KVM/ARM is the mainline Linux hypervisor and supports several cloud deployments of ARM server hardware. More specifically, the contributions of this dissertation include:

1. We identify 28 problematic instructions on ARM that are sensitive and non-privileged and discuss why these instructions cause the ARMv7 architecture to be non-virtualizable.

2. We introduce lightweight paravirtualization to run existing OSes on non-virtualizable ARM systems that do not have hardware support for virtualization. Lightweight paravirtualization is a fully automated script-based technique that modifies guest OS source code to allow building trap-and-emulate hypervisors on ARM.

3. We designed and implemented KVM/ARM, the first hypervisor to leverage ARM hardware virtualization support to run unmodified guest operating systems on ARM multicore hardware.

4. We discuss and analyze the ARM Virtualization Extensions and compare them to equivalent hardware virtualization support provided by Intel for the x86 architecture. This discussion includes an in-depth analysis and comparison of timer and interrupt hardware virtualization support.

5. We introduce split-mode virtualization as a novel approach to hypervisor design, which splits the execution of the core hypervisor across two CPU modes. Split-mode virtualization allows software to leverage specific hardware support in one CPU mode, while at the same time leveraging functionality from an existing operating system running in another CPU mode.

6. We offer advice on how to transfer research ideas into implementations likely to be adopted by an open source community. Our advice is based on experiences building KVM/ARM as

an open-source project from the ground up and later maintaining the open source KVM/ARM implementation for several years.

7. We provide the first measurements of ARM virtualization performance using the ARM Virtualization Extensions on real ARM multicore hardware.

8. We provide the first in-depth performance study comparing both ARM and x86 virtualization using 64-bit ARMv8 server class hardware. We measure the performance of both KVM and Xen using both ARM and x86 and analyze and compare the results.

9. We introduce important microbenchmarks for evaluating virtualization performance and use these to show material differences in performance between ARM and x86 virtualization.

10. We show that application performance in VMs is not necessarily well-correlated with the performance of micro-level hypervisor operations, but also depends largely on other factors such as the overall hypervisor software design support for virtual I/O.

11. We discuss the Virtualization Host Extensions (VHE) which were introduced in ARMv8.1 to better support hosted hypervisor designs based on the experience and performance results of KVM/ARM. We discuss how these extensions can be used to run hosted hypervisors on ARM without using split-mode virtualization, and we show that this alone is not sufficient to significantly improve the performance of hosted hypervisors.

12. We present a new hypervisor design in which a hypervisor and its *hypervisor OS kernel* run in a separate CPU mode from the mode used to run VMs. Modern hypervisors rely on existing full OS kernels to support their hypervisor functionality, and run both the VM kernel and hypervisor kernels in the same CPU mode, resulting in overhead from having to multiplex this CPU mode in software. Our new hypervisor design avoids this overhead by running the hypervisor OS kernel and the VM in completely separate CPU modes.

13. We introduce *el2*Linux, a version of Linux modified to run in ARM's EL2 hypervisor mode. *el2*Linux allows us to run the hypervisor and its OS kernel in a separate CPU mode from the VM kernel.

14. We evaluate the benefits of our new hypervisor design on current ARMv8.0 server class hardware using *el2*Linux, which also provides an early view of the performance benefits of VHE combined with our hypervisor redesign before VHE hardware is available.

15. We show that using *el2*Linux and redesigning the hypervisor to take advantage of running the hypervisor and its OS kernel in a separate mode from the VM results in an order of magnitude improvement for the hypercall microbenchmark result and reduces the virtualization overhead by more than 50% for real application workloads.

Organization of this Dissertation

This dissertation is organized as follows. Chapter 1 discusses why the ARM architecture does not meet the classical definition of virtualizable architectures and shows how lightweight paravirtualization can be used for virtualization on ARM CPU's without hardware virtualization support. Chapter 2 gives an overview of the ARM virtualization extensions, introduces split-mode virtualization, and presents the design and implementation of KVM/ARM. Chapter 3 presents an in-depth performance study of ARM and x86 virtualization. Chapter 4 shows that performance can be improved by running the hypervisor and its OS kernel in a separate CPU mode from the VM and discusses changes to the OS or the hardware along with a redesign of KVM/ARM to take advantage of running in separate CPU modes. Finally, I present some conclusions and directions for future work in Chapter 5.

Chapter 1

ARM Virtualization without Architectural Support

This chapter covers virtualization of ARM without architectural support for virtualization. First, we investigate why the ARM architecture is not virtualizable. The ARM architecture has been revised several times and spans across micro controllers and application CPUs targeting a wide range of computing from cell phones to data center servers. We focus on the last version of the application level architecture, ARMv7-A, before hardware virtualization support was added. While this dissertation mainly focuses on architectural support for virtualization, it is useful to investigate the original premise of ARM virtualization. Next, we present an early prototype of KVM for ARM, which does not rely on hardware virtualization support, but uses lightweight paravirtualization to automatically modify the guest operating system code, allowing the lightly modified guest OS to run in virtual machines on ARMv7 hardware without architectural support for virtualization.

1.1 Requirements for Virtualizable Architectures

Popek and Goldberg's virtualization theorem [66] determines if a particular instruction set architecture (ISA) can be virtualized by a trap-and-emulate hypervisor, which executes most instructions from the VM directly on the real CPU. Architectures satisfying the requirements stated in the theorem are classified as *virtualizable* and those that don't as *non-virtualizable*. The theorem applies to any modern computer architecture, which satisfies a number of properties, such as having at least two separate modes, one privileged and one unprivileged; virtual memory implemented via segmentation or paging; and certain other properties related to management of the processor state. Bugnion et al. [25] cover these properties in length and explain how the properties as they were stated in 1974 can be interpreted and applied to modern architectures, x86, ARM, and MIPS.

If a computer architecture meets the requirements in the theorem, it is possible to construct trap-and-emulate hypervisors, where the virtual machines are "*efficient, isolated, duplicates* of the

real machine". What this means is that the VM executes with only a minor decrease in performance compared to native execution; that the hypervisor remains in control of the real hardware at all times and software in one VM cannot interfere with or access state in another VM, similarly to running on two separate physical machines; and finally, that the VM is sufficiently similar to the underlying machine that software written to run natively on the real machine can also run unmodified in the VM. We refer to these properties of VMs as the **efficiency**, **isolation**, and **equivalence** properties.

The core requirement in the theorem is based on a classification of the instructions in the ISA. **Control-sensitive** instructions are those that can control the configuration of the hardware, for example changing the execution level, disabling interrupts, or configuring the virtual memory settings. **Behavior-sensitive** instructions are those that execute differently depending on the CPU mode they execute in, and the configuration state of the system overall. For example, an instruction that enables or disables virtual memory is control-sensitive, and reading the current CPU mode is behavior sensitive. **Sensitive** instructions are instructions that are either control-sensitive or behavior-sensitive, and instructions that are neither are innocuous. **Privileged** instructions are those that can only be executed from the privileged mode, and, most importantly, trap to privileged mode when executed in non-privileged mode. The theorem then states:

> *For any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.*

That is, we can construct a trap-and-emulate hypervisor on a modern computer architecture, if all sensitive instructions are also privileged. Hypervisors *deprivilege* the guest OS and run the entire VM, including the guest OS written to run in kernel mode, in the unprivileged user mode. All innocuous instructions execute directly on the CPU without intervention of the hypervisor. Sensitive instructions trap, because they are also privileged, to the hypervisor running in kernel mode, and the hypervisor emulates their behavior in software, operating on the virtual machine abstraction instead of the real machine. For example, consider the control-sensitive instruction that disables interrupts. When the guest OS disables interrupts, because the instruction is also privileged, it also traps to the hypervisor, which emulates the behavior by recording in its VM data structures that virtual interrupts should not be signaled to the VM. A guest must not be allowed to disable physical interrupts on the real hardware, because that would violate the VM isolation requirement, and the

hypervisor would lose control of the real machine and other VMs would be affected by no longer seeing any interrupts. As a counterexample, consider a non-virtualizable architecture, in which the disable interrupts instruction is not privileged, but is instead ignored when executed in user mode. In this case, when the guest OS executes the instruction, nothing happens, and the hypervisor will continue to signal virtual interrupts to the VM. An unmodified guest OS will malfunction on such a system, because the hypervisor has no way of realizing that the guest OS has tried to disable interrupts, and the guest OS will receive interrupts when it thinks it has disabled them, for example in critical sections.

1.1.1 Virtualization of ARMv7

This section explains known violations of the requirements for virtualizable architecture of the ARMv7-A [11] and discusses aspects of ARM that are important to someone wishing to build an ARM virtualization system. Many conclusions are also valid for earlier versions of the ARM architecture. We focus our analysis here on the last version of the ARM architecture which was designed before virtualization support was added to the architecture. Virtualization of micro controllers, for example CPUs based on ARMv7-M [14] and ARMv8-M [15], is not considered, because these systems are typically much too resource constrained to run more than a single OS and typically do not have an MMU.

At a high-level from a virtualization perspective, ARM has two *privilege levels*, unprivileged PL0, and privileged PL1. There is a single CPU mode in PL0, user mode, and several CPU modes in PL1. There are actually 6 PL1 CPU modes, one for each type of processor exception, and all with the same privileges and no isolation between each other. The idea behind this design with multiple privileged modes, is that each mode is assigned some private, or *banked*, registers, such as the stack pointer. The stack pointer is banked between each mode, including user mode, which means that reading the stack pointer using the same instruction in any of the 7 CPU modes, user mode and the 6 privileged modes, accesses a different physical register. These 6 privileged mode do not offer any isolation guarantees between each other, but are merely designed for convenience when handling exceptions, and were never used in practice by Linux. Linux instead always immediately changes into one of the 6 privileged modes, *Supervisor* mode, which is the mode used to handle system calls. In the following, we refer to this mode simply as *kernel mode*.

Description	Instructions
Load/Store Multiple (User registers)	LDM (User registers), STM (User registers)
Status registers	CPS, MRS, MSR, RFE, SRS, LDM (Exception return)
Data processing	ADCS, ADDS, ANDS, BICS, EORS, MOVS, MVNS, ORRS, RSBS, RSCS, SBCS, SUBS
Load/Store Unprivileged	LDRBT, LDRHT, LDRSBT, LDRSHT, LDRT STRBT, STRHT, STRT

Table 1.1: ARMv7 sensitive non-privileged instructions

Description	Instructions
Load/Store Multiple (User registers)	LDM (2), STM (2)
Status registers	CPS, MRS, MSR, RFE, SRS, LDM (3)
Data processing	ADCS, ADDS, ANDS, BICS, EORS, MOVS, MVNS, ORRS, RSBS, RSCS, SBCS, SUBS
Load/Store Unprivileged	LDRBT, LDRT STRBT, STRT

Table 1.2: ARMv6 sensitive non-privileged instructions

The ARM architecture does not satisfy the requirements for a virtualizable architecture. It contains sensitive, unprivileged instructions, which violate the Popek and Goldberg's virtualization theorem. For ARMv7, we identify 28 problematic instructions that are sensitive and unprivileged. Table 1.1 shows the specific instructions identified for ARMv7. Many of these are also present in earlier versions of the architecture. For example, Table 1.2 shows the specific instructions identified for ARMv6. I compiled the table by simply going over every instruction in the ARM Architecture Reference Manual [11] (ARM ARM), which is the complete specification of the ISA, and examining the usage restrictions for each instruction. The instructions can be categorized as instructions that operate on user mode registers, access status registers, and perform memory accesses that depend on CPU mode.

LDM (User registers) loads multiple user mode registers from memory and STM (User registers) stores multiple user mode register to memory. This means, where a normal LDM or STM instruction accesses a consecutive memory area and transfers data to/from registers, using the registers specific to the current CPU mode, the (User registers) version will always use user mode registers. For example, user mode and kernel mode have separate stack pointers, and using a normal LDM instruction which loads the stack pointer in kernel mode will load the kernel mode stack pointer. In contrast, using the LDM (User registers) version in kernel mode will instead load the

user mode stack pointer. A hypervisor deprivileges kernel execution and runs the kernel in user mode, and therefore must multiplex the single user mode stack pointer register between the virtual user and virtual kernel modes. However, since the Load/Store Multiple (User registers) instructions allow access to user mode registers, the hypervisor must emulate such accesses when running the guest OS kernel, because the user mode register will only exist in a hypervisor managed data structure at that time. This means that Load/Store Multiple (User registers) is a behavior-sensitive instruction, because its semantics depends on the current CPU mode. The ARM architecture defines execution of the Load/Store Multiple (User registers) instruction as *unpredictable* in user mode. Unpredictable is an architectural concept, which has specific constraints for a given implementation of the architecture, but a common valid interpretation of this concept is that the instruction is simply ignored in user mode. Other versions of the architecture than ARMv7 define these instructions as executing normally without generating a trap. Unfortunately, in all cases, Load/Store Multiple (user registers) is not a privileged instruction, and violates the requirement for a virtualizable architecture.

Status register instructions access special ARM status registers, the Current Program Status Register (CPSR) and Saved Program Status Register (SPSR). CPSR specifies the current mode of the CPU and other state information, some of which is privileged. SPSR is a banked register available in all privileged modes except *System* mode. The basic ARM protection mechanism works by copying the CPSR to the SPSR when the processor enters the respective privileged mode via an exception so that CPU state information at the time of the exception can be determined. Similarly, returns from exceptions are performed by copying the SPSR to the CPSR and changing the program counter (PC) in a single operation. The status register instructions are all sensitive and non-privileged and explained in more detail in Table 1.3.

The data processing instructions are special versions of normal data processing instructions which also perform an exception return. They replace the CPSR with the SPSR in addition to performing a simple calculation on their operands. These instructions are denoted in the ARM assembly language by appending an 'S' to the normal instruction mnemonic. They are control-sensitive because they change the CPU mode, and they are non-privileged because they are unpredictable when executed in user mode.

Load/Store Unprivileged access memory with the same access permissions as unprivileged memory accesses from user space would have. The virtual memory system on ARM processors

Instruction	Explanation
CPS	Change Process State changes privileged fields in the CPSR, and is therefore control-sensitive. CPS is ignored by the processor if executed in user mode, and therefore non-privileged.
MRS	Move to Register from Banked or Special register moves a value from a banked register or from the SPSR into a general purpose register. MRS is behavior-sensitive because the result of its execution depends on the current CPU mode, and is non-privileged because it is unpredictable in user mode.
MSR	Move to Banked or Special register from a general purpose register or an immediate value. It is both control-sensitive because it can modify privileged fields in the CPSR, and behavior-sensitive because it only updates a subset of the fields in the CPSR accessible in kernel mode when run in user mode. It is non-privileged because it is unpredictable in user mode if writing the SPSR and silently changes behavior when writing the CPSR, but does not trap.
RFE	Return From Exception loads the PC and the CPSR from memory. It is control-sensitive because it changes privileged state and non-privileged because it is unpredictable in user mode.
SRS	Store Return State stores the LR and SPSR of the current mode to the stack of a specified mode. It is behavior-sensitive because its execution depends on the current mode and non-privileged because it is unpredictable in user mode.
LDM (Exception return)	Load Multiple (Exception return) loads multiple registers from consecutive memory locations and copies the SPSR of the current mode to the CPSR. It is both control-sensitive because it affects privileged state and behavior-sensitive because its execution depends on the current mode. It is non-privileged because it is unpredictable in user mode.

Table 1.3: Sensitive non-privileged status register instruction descriptions

uses access permissions to limit access to memory depending on the CPU mode. This means that virtual memory regions can be configured to be accessible only in kernel mode and accesses from user space generate a trap to kernel mode. Load/Store Unprivileged are used to perform accesses from kernel mode which generate the same traps as an access to the same memory region would have if executed in user mode. This can be useful for an OS kernel to validate user space accesses to memory. However, when executed in user mode, these instructions behave as regular memory access instructions. A memory access from kernel mode to a memory region only accessible by kernel mode should succeed with a normal load/store instruction, but a load/store unprivileged would generate an exception (permission fault) on the memory region. A hypervisor would have no way

to emulate this difference, because neither normal load/store nor load/store unprivileged instructions trap, and they have the same exact semantics when executed in user mode. These instructions are behavior-sensitive, because they execute differently depending on the current CPU mode, and non-privileged because they don't trap when executed in user mode.

1.1.2 Virtualization of ARMv8

ARMv8 changed both the nomenclature and protection model and instead of defining several CPU modes, the CPU can be at one of several *Exception Levels*, EL0 is the least privileged level, and EL3 is the most privileged level. EL0 is used for user applications, EL1 is used for a kernel, EL2 is used to run a hypervisor, and EL3 is used to run a secure monitor, which can switch between secure and non-secure execution. While ARMv8 has optional hardware virtualization support, ARMv8 without these hardware features is easier to virtualize than ARMv7. The intention when designing ARMv8 was to make the architecture satisfy the requirements for a virtualizable architecture [69]. There are no load/store multiple instructions on ARMv8. The status register instructions have been rewritten and any instruction accessing privileged state or affected by the current CPU mode can either be configured to trap, or always traps, to EL1 when executed in EL0. The special versions of the data processing instructions are no longer available. Unfortunately, the *Load/Store Unprivileged* instructions have the same semantics as on ARMv7 (explained above), and therefore still violate the virtualization criteria. This is the only violation of the virtualization criteria we have identified for ARMv8.

1.2 KVM for ARM without Architectural Support

This section presents our work on an early prototype of KVM for ARM without architectural support for virtualization. The prototype is built for ARMv6 and ARMv7. KVM is a hosted hypervisor integrated into Linux and was originally designed specifically for architectures that had hardware support for virtualization. Although ARM originally did not have such architectural support, the dominance of Linux on ARM made KVM an attractive starting point for building an ARM hypervisor. Linux supports a wide range of ARM hardware and provides basic OS functionality such as scheduling and memory management. KVM for ARM simplified the development of an ARM

hypervisor by combining the benefits of lightweight paravirtualization with existing Linux infrastructure support for ARM.

As established above in Section 1.1.1, ARM does not satisfy the requirements for a virtualizable architecture and therefore does not support running unmodified operating systems using trap-and-emulate. There are two known ways to address this problem, binary translation and paravirtualization. Binary translation is difficult to implement efficiently, requires significant engineering efforts to implement correctly, and is likely to introduce overhead on ARMv7 systems that are relatively weaker than x86 systems. A key advantage of binary translation is its ability to run unmodified closed-source guest OSes, which was a very important use case for x86 with its widespread deployment of Microsoft Windows. For ARMv7, however, deployments of software stacks typically involve compiling a custom OS kernel from source code for the specific device and compatibility with binary OS images is of less concern.

Paravirtualization as a term was introduced by the work on the Denali [83, 82] isolation kernel and was later popularized by Xen [16]. The basic idea behind paravirtualization is to sacrifice compatibility with the underlying hardware architecture in favor of a custom virtual machine interface defined by the hypervisor. By defining a custom VM interface, the hypervisor system is no longer bound by limitations of the underlying hardware and the interface can be designed to optimize certain aspects of the virtualized system such as scalability and performance. The downside is that software designed to run on real hardware systems cannot run inside the VM unless modified to work with the new interface.

Paravirtualization changes the guest OS to work with its custom VM interface. Typically, a hypervisor will expose higher level functionality through hypervisor calls, or *hypercalls*, available from the VM to the hypervisor. Similar in concept to system calls, the VM can request certain functionality from the hypervisor, such as mapping memory. These calls are used by the guest OS to ensure correct functionality. Instead of relying on trap-and-emulate on all sensitive instructions, functionality in the guest OS that uses sensitive instructions is rewritten to make hypercalls to directly request emulation. A clear benefit is that where a trap-and-emulate hypervisor potentially traps multiple sensitive operations when the guest OS performs some operation, a paravirtualized guest OS can collapse all these traps into a single hypercall. A drawback is that the hypervisor calls are often much more complicated to implement than emulating a simple instruction. The

complexity challenges when implementing hypervisor call handlers can lead to serious security vulnerabilities [84, 85, 86]. Traditional paravirtualization also requires detailed understanding of the guest operating system kernel to know how to modify its source code, and then requires ongoing maintenance and development of paravirtualized guest OSes. Furthermore, the modifications are both architecture and operating system dependent, which adds to the maintenance burden.

Xen uses paravirtualization to support VMs running full OSes on the non-virtualizable x86 IA-32 ISA. They accomplish this by designing the custom VM interface to be similar to the underlying x86 IA-32 interface, but with modifications to support running in a VM. For example, they designed specific hypercalls to handle interrupts and designed a Xen-specific virtual MMU architecture to support virtual memory in VMs so that the guest OS doesn't manage its own page tables and perform TLB maintenance operations, but instead simply asks the hypervisor to allocate and free memory on its behalf.

1.2.1 Lightweight Paravirtualization

As an alternative approach to traditional paravirtualization we introduced *lightweight paravirtualization* [34]. Lightweight paravirtualization is a script-based method to enable a trap-and-emulate virtualization solution by automatically modifying the source code of a guest OS kernel to issue calls to the hypervisor instead of issuing sensitive instructions. Lightweight paravirtualization is architecture specific, but operating system independent. It is completely automated and requires no knowledge or understanding of the guest operating system code.

Lightweight paravirtualization differs from Xen's paravirtualization by replacing individual instructions instead of entire functions in the guest OS. Lightweight paravirtualization requires no knowledge of how the guest is engineered and can be applied automatically on an OS source tree. As an example of how these approaches differ, an early prototype of Xen on ARM [47] based on traditional paravirtualization defines a whole new file in the Linux kernel's memory management code [1], which contains functions based on other Xen macros to issue hypercalls to manage memory. Existing kernel code is modified to conditionally call these Xen functions in many places throughout the guest kernel code. Lightweight paravirtualization, on the other hand, completely maintains the

[1] `arch/arm/mm/pgtbl-xen.c`

original kernel logic, which drastically reduces the engineering cost and makes the solution more suitable for test and development of existing kernel code.

However, there is a performance trade-off between lightweight paravirtualization and traditional paravirtualization. The former simply replaces sensitive instructions with traps to the hypervisor, so that each sensitive instruction now traps and is emulated by the hypervisor. If there are many sensitive instructions in critical paths, this can result in frequent traps and if traps are expensive, this can become a performance bottleneck. Traditional paravirtualization approaches typically optimize this further by replacing sections of code that may have multiple sensitive instructions with one paravirtualized hypercall to the hypervisor instead of repeated traps on each individual sensitive instruction, thereby improving performance.

Lightweight paravirtualization can be a more favorable approach in the context of supporting virtualization on ARM platforms without virtualization support compared to Xen's paravirtualization approach for x86. One idea behind using lightweight paravirtualization, which was shared by the designers of VMware's Mobile Virtualization (MVP) [17] system, is that the trap cost on ARMv7 and earlier is lower than on x86, and that ARM's sensitive instructions occurred less frequently in critical code paths compared to x86 code. The ARM ISA also makes it significantly easier to replace individual instructions in the instruction stream compared to IA-32, because ARM mostly uses fixed-width instructions. (Thumb-2 code uses a mix of 16-bit and 32-bit instructions.) These architectural differences between ARM and x86 make lightweight paravirtualization particularly attractive on ARM.

Let us take a closer look at how lightweight paravirtualization works. To avoid the problems with sensitive non-privileged instructions, lightweight paravirtualization is used to patch the guest kernel source code and replace sensitive instructions with instructions generating a trap. There is no need to worry about user space software as user space applications will execute in the CPU mode they were designed for, similarly to native execution. Sensitive instructions are not generated by standard C-compilers and are therefore typically only present in assembler files and inline assembly. KVM for ARM's lightweight paravirtualization is done using an automated scripting method to modify the guest kernel source code. The script is based on regular expressions and has been tested on a number of Linux kernel versions. The script supports inline assembler syntax, assembler as part of preprocessor macros, and, assembler macros. As an alternative approach, with knowledge

of the compiled format of guest kernel binaries, a tool could automatically scan the text section of the guest kernel binary and patch all sensitive instructions. This approach covers relocatable binaries as the text before relocation would have been patched already. However, the approach has two potential limitations. First, self-modifying code which generates encodings of sensitive instructions are not supported. Second, executing modules or other kernel mode executables at run time would require first processing the binaries. In contrast to binary translation, lightweight paravirtualization statically replaces sensitive instructions before the instructions are loaded on a simple 1:1 basis. Efficient binary translation techniques change the instruction stream at run time and rewrites basic blocks of execution into linked translated blocks for optimal performance [1], a much more complicated operation.

Sensitive non-privileged instructions are replaced with trap instructions and KVM for ARM emulates the original sensitive instruction in software when handling the trap. However, the hypervisor must be able to retrieve the original sensitive instruction including its operands to be able to emulate the sensitive instruction when handling a trap. For example, if the sensitive instruction was the MRS instruction, which moves from a status register to a GP register, the instruction will include in its operands the destination GP register. KVM for ARM therefore defines an encoding of all the sensitive non-privileged instructions and their operands into privileged instructions, so that the privileged instruction can be read back by the hypervisor when handling a trap, the hypervisor can decode the trapped privileged instruction, and reconstruct the original sensitive instruction and its operands.

The challenge is to find enough privileged instructions in the ISA which are not already used by a guest kernel and have unused bit fields in the instructions that can be used to encode the sensitive instructions. The ARM system call instruction, `SWI`, always traps, and contains a 24-bit immediate field (the payload), which can be used to encode sensitive instructions., but this is not quite sufficient to encode all the sensitive non-privileged and their operands. Some additional instructions, that trap, are needed for the encoding.

The ARMv7-A ISA also defines privileged *coprocessor access* instructions which are used to access the *coprocessor interface*. This interface does not relate to any actual physically separate processor, but is merely used to extend the instruction set by transferring data between general purpose registers and registers belonging to one of the sixteen possible coprocessors. For example, the

ISA reserves coprocessor number 15, the *system control coprocessor*, to control the virtual memory system. Coprocessor access instructions are sensitive and privileged — and always trap. Luckily, using a coprocessor number which is unused in the ISA always generates an undefined exception and traps to kernel mode. KVM for ARM re-purposes the unused coprocessor access instruction space for encoding sensitive non-privileged instructions. Specifically, coprocessor numbers zero through seven are not used in any known implementation, and KVM for ARM uses the MRC and MCR instructions with those coprocessor numbers. These instructions have 24 bits of operands, and, regardless of the operands used, the instructions always trap when accessing the unused coprocessor numbers and can therefore be leveraged to encode the sensitive non-privileged instructions. In combination, the `SWI` and coprocessor access instructions are sufficient to encode all possible sensitive non-privileged instructions and their operands.

The VMM must be able to distinguish between guest system calls and traps for sensitive instructions. KVM for ARM assumes that the guest kernel does not make system calls to itself. Under this assumption, if the virtual CPU is in privileged mode, the payload is simply interpreted and the original sensitive instruction is emulated. We are not familiar with any open source ARM OS which makes system calls to itself using specific values in the immediate field. Furthermore, this is only a concern when the virtual CPU is in kernel mode, not when it is in user mode, because user when the `SWI` instruction is executed from the VM's user mode it is simply handled as a system call from guest user space to the guest kernel. The hypervisor emulates this transition in software by storing the hardware user mode context in memory and changing the hardware user mode context to guest kernel context.

As discussed in Section 1.1.1, ARMv7 defines 28 sensitive non-privileged instructions in total. KVM for ARM encodes the instructions by grouping them in 16 groups; some groups contain many instructions and some only contain a single instruction. The upper 4 bits in the `SWI` payload indexes which group the encoded instruction belongs to (see Table 1.4). This leaves 20 bits to encode each type of instruction. Since there are 5 status register access functions and they need at most 17 bits to encode their operands, they can be indexed to the same type and be sub-indexed using additional 3 bits. There are 12 sensitive data processing instructions and they all use register 15 as the destination register and they all always have the S bit set (otherwise they are not sensitive). They are indexed in two groups: one where the I bit is set and one where it's clear. In this way, the

Index	Group / Instruction
0	Status register access instructions
1	LDM (User registers), P-bit clear
2	LDM (User registers), P-bit set
3	LDM (Exception return), P-bit clear and W-bit clear
4	LDM (Exception return), P-bit set and W-bit clear
5	LDM (Exception return), P-bit clear and W-bit set
6	LDM (Exception return), P-bit set and W-bit set
7	STM (User registers), P-bit set
8	STM (User registers), P-bit clear
9	LDRBT, I-bit clear
10	LDRT, I-bit clear
11	STRBT, I-bit clear
12	STRT, I-bit clear
13	LDRHT, STRHT, I-bit clear
14	Data processing instructions, I-bit clear
15	Data processing instructions, I-bit set

Table 1.4: Sensitive instruction encoding types

data processing instructions need only 16 bits to encode their operands leaving 4 bits to sub-index the specific instruction out of the 12 possible. The sensitive load/store multiple and load/store with translation instructions are using the remaining index values as can be seen in Table 1.4.

In Table 1.4 only the versions of the load/store instructions with the I-bit clear are defined. This is due to a lack of available bits in the `SWI` payload. The versions with the I-bit set are encoded using the coprocessor access instruction. When the I-bit is set, the load/store address is specified using an immediate value which requires more bits than when the I-bit is clear. Since the operands for coprocessor access instructions use 24 bits, 3 bits can be used to distinguish between the 5 sensitive load/store instructions. That leaves 21 bits to encode the instructions with the I-bit set, which is sufficient for all immediate encodings and options.

An example may help illustrate the implementation of the KVM for ARM solution. Consider this code in `arch/arm/boot/compressed/head.S`:

```
mrs     r2, cpsr    @ get current mode
tst     r2, #3      @ not user?
bne     not_angel
```

The `MRS` instruction in line one is sensitive, since when executed as part of booting a guest, it will simply return the hardware CPSR. However, KVM for ARM must make sure that it returns the

virtual CPSR instead. Thus, it can be replaced with a `SWI` instruction as follows:

```
swi     0x022000    @ get current mode
tst     r2, #3      @ not user?
bne     not_angel
```

When the `SWI` instruction in line one above generates a trap, KVM for ARM loads the instruction from memory, decodes it, emulates it, and finally returns to the next instruction.

1.2.2 CPU Exceptions

An exception to an ARM processor can be generated synchronously from executing an instruction, for example a data abort is generated if a virtual address lookup results in a page fault, and can be generated asynchronously if an input pin is asserted on the processor, for example because a hardware interrupt occurs. Synchronous exceptions are known as traps and asynchronous exceptions are commonly referred to as interrupts, although the ARM architecture defines three types of asynchronous interrupts. All exceptions halt current execution and jumps to an *exception vector*. The hardware also changes from non-privileged to privileged mode if executing at a non-privileged mode when the exception occurs. The *exception vector* is a fixed offset defined per exception type from a *vector base address register* (VBAR) on newer versions of the ARM architecture and from one of two fixed locations (the low and high vectors) on ARMv5 and older.

To maintain the VM isolation requirement and remain in control of the hardware, an ARM hypervisor must install special code in the page mapped at the exception vector to be able to intercept exceptions and either return control to the hypervisor or emulate exceptions for the VM when, for example an interrupt occurs. But, VMs will expect to be able to use the entire virtual address space defined by the architecture, and therefore there is a potential conflict between the VM and the hypervisor to reserve the exception page for each their own use.

The ARMv5 through ARMv7 architecture versions define seven exceptions (not including exceptions introduced as part of the security or virtualization extensions) shown in Table 1.5. The reset exception occurs when the physical reset pin on the processor is asserted. The behavior is similar to power cycling the processor. Undefined exceptions happen when an unknown instruction is executed or when software in user mode tries to access privileged coprocessor registers or when software accesses non-existing coprocessors. Software Interrupt happens when SWI instructions

Exception	Offset
Reset	0x0
Undefined	0x4
Software Interrupt	0x8
Prefetch Abort	0xc
Data Abort	0x10
IRQ	0x18
FIQ	0x1c

Table 1.5: ARMv7-A Exceptions

are executed. Prefetch and data abort exceptions happen when the processor cannot fetch the next instruction or complete load/store instructions, respectively. For example, prefetch and data aborts can be caused by missing page table entries or by memory protection violations. IRQs and FIQs are caused by external hardware asserting a pin on the processor.

ARM operating systems must configure the exception environment before any exceptions occur. ARM processors power on with interrupts disabled, and the OS avoids generating traps before configuring an exception handler to ensure a valid boot process. Typically, once the OS configures page tables and enables virtual memory, it maps a page at the exception vector base address and installs instructions to jump to specific exception handlers in that page. Only then can the operating system enable interrupts and generate traps and handle them. However, the VM cannot be in control of the exception vectors, because it could then prevent the hypervisor from gaining control of the system, violating the isolation requirement. Instead, KVM for ARM installs a set of hypervisor specific exception handlers on a new page, and maps this page at the exception vector base address when running the VM. The hypervisor exception handlers return to the hypervisor execution environment and processes the exception to either service the VM or the underlying hardware. We note that the VM kernel does not handle hardware exceptions directly, but only sees virtual interrupts generated and emulated by the hypervisor.

Since the hypervisor must control the exception vector page, this virtual address is not available to the guest. KVM for ARM solves this by simply moving the exception vector to an address not currently in use by the guest if the guest tries to access the exception page. On versions of the architecture before ARMv7 where the vector base address cannot be freely configured, the architecture provides a bit to let software choose between placing the exception vector at one of two

predefined addresses. KVM for ARM can simply switch between these two addresses as needed for any guest that accesses the exception vector page. The exception handlers used by KVM for ARM are compiled as a part of KVM and are relocated to a hypervisor exception page, which can be used when running VMs.

1.2.3 Memory Virtualization

Virtual memory is already a property of the hardware, but virtual memory is managed by the memory management unit (MMU) on modern architectures. To maintain the VM equivalence property, hypervisors must properly virtualize the MMU and multiplex the MMU functionality between multiple VMs. Often, this is referred to as memory virtualization in the context of running virtual machines, however ambiguous the term is.

Unmodified guest OSes written to manage virtual and physical memory on a real hardware system must also execute correctly in the functionally equivalent VM. The guest OS must also be protected from guest applications, similar to real hardware. Second, the hypervisor must be protected from the guest OS and applications to satisfy the isolation VM property. KVM for ARM uses a well-known technique from the x86 world known as *shadow page tables* [24] which leverages the hardware MMU to virtualize memory for the VM.

Since the VM is isolated from the underlying physical machine, and the physical memory resource is multiplexed between multiple virtual machines, the guest's view of physical memory is entirely separate from the actual physical memory layout. The guest view of physical memory is called the Guest Physical Address (GPA) space. The guest operating system maps from Virtual Addresses (VAs) to GPAs using its own page table structures, but these page tables are not used by the physical MMU. Instead, the hypervisor creates shadow page tables, which translate from the guest VAs to real Physical Addresses (PAs). Figure 1.1 illustrates the various address spaces and their relationship.

Guest pages are mapped using the shadow page tables on demand, when the hypervisor handles page faults from VMs. An entry is created in the shadow page table for the faulted virtual address. Shadow page table entries translate directly from VAs to PAs. The VA is translated to a GPA by walking guest page tables in software. KVM for ARM can translate any GPA into a PA by using its own data structures. Each shadow page table entry is therefore the result of combining two

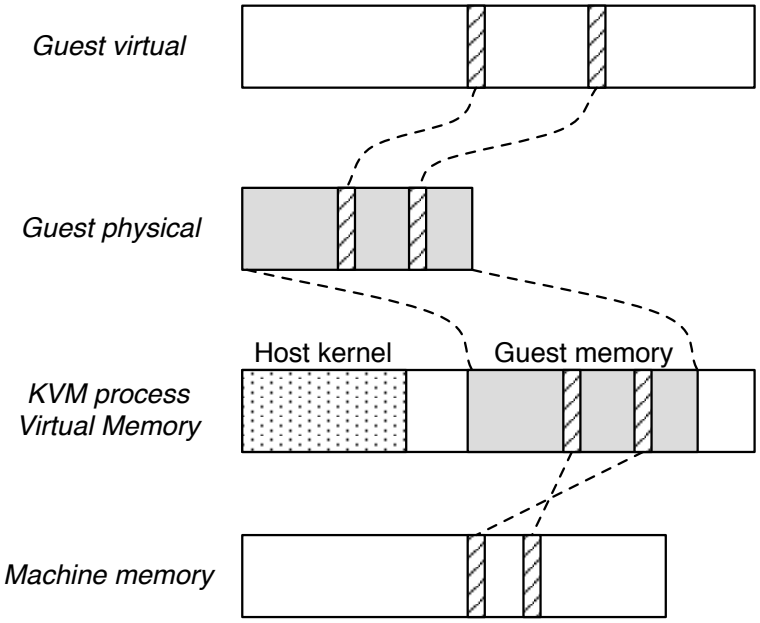

Figure 1.1: Address Space Mappings

translation stages into a single stage, and translates directly from VAs to PAs and is then walked by the hardware MMU when running the VM.

Two special entries are always mapped by the hypervisor in the shadow page tables and reserve part of the guest's VA space, the *hypervisor exception page* and the *shared page*. When the hypervisor enters the VM, the entire virtual address space is completely changed from the hypervisor and host OS's view to the guest OS's view. Both the virtual address space and registers are changed when moving from host to VM execution context, and these two contexts are essentially in two different worlds. Therefore, we refer to the transition between the hypervisor and the VM as a *world switch*. However, when moving from one page table to the other, the ARM architecture requires that this is done from a page which is shared between the two page tables. KVM for ARM therefore also has to support a *shared page*, which is a page mapped at the same VA in both the host and VM's virtual address space. The shared page contains two sequences of code, one to move from the hypervisor to the VM, and one to move from the VM to the hypervisor, and also includes a small amount of data, for example to store the host's page table base address, which is used to return to the host OS from the VM. If the VM tries to write to any of these two pages it will generate a permission fault to the hypervisor and the hypervisor moves the location of the special page and simply chooses a new address in the guest's VA space for the special page.

The hypervisor must keep the shadow page tables up to date with any changes to either of the two translation stages combined into the shadow page table entries. GPA to PA translations are maintained by the hypervisor itself and are therefore trivial to propagate into the shadow page table entries, but changes to guest page tables are more difficult to track. For example, if the guest kernel uses copy-on-write, it will modify its own page table mappings on the first write to a COW memory region. Fortunately, such a change in address mappings requires the guest to also invalidate any TLB entries potentially caching the old mapping, and TLB invalidation is a privileged operation that traps to the hypervisor. The hypervisor therefore detects that the guest has modified all or part of its address space and invalidates corresponding shadow page table entries as needed.

Since the guest OS must run in the non-privileged user mode to protect the hypervisor and the rest of the system, and since ARMv7 allows a page to have different permissions for privileged and non-privileged accesses, KVM for ARM must virtualize the access permissions. For example, if the guest page table entry for a particular page allows access in privileged mode but no access in user mode, and the VM is running in virtual kernel mode (but actually running in user mode), then the shadow page table permissions must allow non-privileged access. When the VM changes its emulated mode to user mode, the shadow page tables must not allow any accesses. The hypervisor therefore has to update access permissions on the shadow page table when the virtual CPU changes its virtual mode between privileged and non-privileged execution and vice versa.

ARMv7 and earlier architecture revisions define *domains*, a mechanism to overwrite page access controls. Domains configure 1 MB virtual address regions to either be completely non-accessible, follow access permissions in the page tables, or always allow full read/write access regardless of the CPU mode. If the guest configures a region that covers one of the two special pages, the shared page or the exception page, to allow all accesses to the region, then the hypervisor must reconfigure the domain of that region in the shadow page tables to instead use the access permissions in the page table, and configure the special pages to only allow privileged access and configure the remaining guest pages for non-privileged access.

1.3 Related Work

A number of early efforts explored virtualization on ARM. An early effort to port Xen to ARM [47] supported ARM CPUs without hardware virtualization support and used paravirtualization and significantly modified the guest OS to run on the non-virtualizable ARM architecture. This type of paravirtualization requires intimate knowledge of the guest OS source code and is difficult to maintain. For example, this particular attempt at ARM virtualization modified more than 4500 lines of code by hand in Linux, and these changes were never supported in mainline Linux. Eventually this effort never materialized with a healthy development ecosystem, no guest OS other than a Linux 2.6.11 kernel was supported, and the effort was abandoned in favor of a new Xen implementation for ARM. In contrast to the classic full paravirtualization approach, we introduced lightweight paravirtualization which requires no knowledge of the guest operating system, is fully automated, and can be applied to any guest operating system with access to its source code or binary executable image format. While our early implementation of KVM for ARM did not get directly merged upstream in Linux or was commercially deployed, the code base was used as a foundation for what eventually became KVM/ARM, the Linux ARM hypervisor. This was in large part due to KVM for ARM initially being based on Linux, and being able to leverage the Linux kernel development community early in the process.

Heiser and Leslie [44] discuss the differences between hypervisors and microkernels based on their experiences building OKL4, a microkernel and hypervisor hybrid for ARM. Their work is done before hardware virtualization support was added to the ARM architecture, and is based on some form of paravirtualization. Unfortunately, not many details regarding their approach to ARM virtualization is presented, but instead their work focuses on theoretically comparing the functionality and abstractions provided by microkernels and hypervisors. OKL4 specifically targeted the embedded market and required custom modified guest operating systems. In contrast, our work on KVM for ARM focused on providing a general purpose hypervisor with few, simple, and automated modifications to guest operating systems. Additionally, our work focused on the interaction between the hypervisor software and the computer architecture, where their work focused on the relationship between hypervisors and microkernels.

The VMware Mobile Virtualization Platform (MVP) [17] also used lightweight paravirtualiza-

tion to support virtual machines on ARM processors without hardware support for virtualization. Their hosted hypervisor design resembled the VMware workstation design [77] in which the VMM executes in a separate environment from the host kernel and only loads a small driver in the host OS being able to switch between the VMM and host execution contexts and interface with support from the host OS, for example from hardware drivers. Additionally, the authors focus on a very specific use case, the Bring Your Own Device (BYOD) model where VMs provide separate mobile computing environments on a phone. For example, they provide an analysis of the best file systems to use for NAND flash memory and SD cards typically present in mobile phones, and how to support network and telephony in a virtual mobile phone environment. In contrast, our early work on KVM for ARM focuses on the general challenges in providing virtualization on ARM without architectural support for virtualization. Furthermore, KVM for ARM was integrated with the host kernel and did not have to construct a separate execution environment for the VMM.

Cells [6, 30] used operating system virtualization similar to containers to support Virtual Phones (VPs) using kernel namespaces. While Cells did not support virtual machines in the classic sense, it provided very efficient virtualization on ARM processors without hardware virtualization support. Furthermore, Cells benefited from its integration with Linux and device drivers to provide a smooth user experience with fully accelerated graphics in all VPs, something which was not supported by the commercial MVP approach. Where our work on KVM focused on providing a general purpose hypervisor that provided full system virtualization, our work on Cells focused on providing multiple virtual phone environments with full device support on real smartphones with as little performance overhead as possible.

Ding et al. [39] expanded on our work and used the same techniques as presented in this dissertation, but further optimized the implementation and presented some performance results. Some commercial virtual solutions for ARM CPUs without architectural support for virtualization [42, 67] are also available, primarily targeting the embedded market, but little is known about their specific designs or experiences.

Pinto et al. [65] describe how to run multiple instances of custom OSes using ARM TrustZone technology on ARM systems without architectural support for virtualization. Their solution statically partitions the system, and does not offer flexible assignment of resources such as memory, which is typically provided by virtualization solutions. CPU, memory, and I/O devices are not vir-

tualized and multiplexed between VMs, but are statically partitioned at boot time and assigned to specific software instances. Their solution requires enlightened guests that are aware of a partitioned physical memory map, and they rely on an optional peripheral, a TrustZone address space controller, to ensure memory isolation between multiple VMs.

1.4 Summary

In this chapter, I briefly presented the requirements for virtualizable architectures and analyzed why ARMv7 does not meet those requirements. We identified 28 sensitive non-privileged instructions on ARMv7, most of which are also present in earlier versions of the architecture. We also briefly discussed virtualization of ARMv8 without hardware virtualization support. ARMv8 is mostly virtualizable, with the exception of a number of Load/Store Unprivileged instructions, which are still behavior-sensitive and not privileged.

I then presented an early prototype of KVM for ARM without architectural support for virtualization. KVM for ARM uses lightweight paravirtualization to statically patch sensitive non-privileged instructions in the guest kernel, replacing the sensitive instructions with privileged instructions that trap to the hypervisor. KVM for ARM defines an encoding of all sensitive non-privileged instructions into unused parts of the ISA, which are still interpreted as privileged instructions and trap to kernel mode when executed in user mode. I also presented the challenges in dealing with CPU exceptions and virtualizing the MMU on ARMv7.

Chapter 2

The Design and Implementation of KVM/ARM

ARM-based devices are dominating the smartphone, tablet, and embedded markets. While ARM CPUs have benefited from their advantages in power efficiency and cost in these markets, ARM CPUs also continue to increase in performance such that they are now within the range of x86 CPUs for many classes of applications. The introduction of powerful ARMv7 CPUs and the 64-bit ARMv8 architecture is spurring an upward push of ARM CPUs into traditional server systems and a new class of network infrastructure devices. 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 and networking, ARM has introduced hardware virtualization support into their architecture.

However, there are important differences between ARM and x86 architectural support for virtualization. ARM designed their virtualization support around a new CPU mode, designed to run the hypervisor, but not designed to run a full OS or existing OS kernels. As a result, popular Type 2 hypervisor designs on x86 are not directly amenable to ARM. x86 duplicates the entire state of the CPU between two modes of operation, orthogonal to the CPU modes, both capable of running full OSes. Unlike x86-based systems, there is no PC-standard hardware equivalent for ARM. The ARM market is fragmented with many different vertically integrated ARM platforms with non-standard hardware. Virtualizing ARM in a manner that works across the diversity of ARM hardware in the absence of any real hardware standard is a key challenge.

This chapter introduces split-mode virtualization, a new approach to hypervisor design that splits the core hypervisor so that it runs across different privileged CPU modes to take advantage of the specific benefits and functionality offered by each CPU mode. This approach provides key benefits in the context of ARM virtualization. ARM introduced a new CPU mode for running hypervisors called Hyp mode, but Hyp mode has its own set of features distinct from existing kernel modes. Hyp mode targets running a standalone hypervisor underneath the OS kernel, and

was not designed to work well with a hosted hypervisor design, in which the hypervisor is integrated with a host kernel. For example, standard OS mechanisms in Linux would have to be significantly redesigned to run in Hyp mode. Split-mode virtualization makes it possible to take advantage of the benefits of a hosted hypervisor design by running the hypervisor in normal privileged CPU modes to leverage existing OS mechanisms without modification while at the same time use Hyp mode to leverage ARM hardware virtualization features.

We designed and implemented KVM/ARM, not just to build a research prototype, but to create a platform that would be widely used in practice and therefore could serve as a foundation for future research by the wider research community as well as for production use. A major design consideration in our work was ensuring that the system would be easy to maintain and integrate into the Linux kernel. This is especially important in the context of ARM systems which lack standard ways to integrate hardware components, features for hardware discovery such as a standard BIOS or PCI bus, and have no standard mechanisms for installing low-level software. A standalone bare metal hypervisor would need to be ported to each and every supported hardware platform, a huge maintenance and development burden. Linux, however, is supported across almost all ARM platforms and by integrating KVM/ARM with Linux, KVM/ARM is automatically available on any device running a recent version of the Linux kernel. However, due to the differences between ARM and x86 architectural support for virtualization, KVM cannot just simply be ported from x86 to ARM, but requires a new hypervisor design, split-mode virtualization.

The initial implementation of KVM/ARM added less than 6,000 lines of ARM code to Linux, a much smaller code base to maintain than standalone hypervisors. KVM/ARM for 32-bit ARMv7 was accepted as the ARM hypervisor of the mainline Linux kernel as of the Linux 3.9 kernel and the 64-bit ARMv8 support was added in Linux 3.10, ensuring its wide adoption and use given the dominance of Linux on ARM platforms. KVM/ARM [56, 36] is the first hypervisor to leverage ARM hardware virtualization support to run unmodified guest operating systems on ARM multi-core hardware. Based on our open source experiences, we offer some useful hints on transferring research ideas into implementations likely to be adopted by the open source community.

We demonstrate the effectiveness of KVM/ARM on the first publicly available ARMv7 hardware with virtualization support. The results are the first published measurements of a hypervisor using ARM virtualization support on real hardware. We compare against the standard widely-

used Linux KVM x86 hypervisor and evaluate its performance overhead for running application workloads in virtual machines (VMs) versus native non-virtualized execution. The results show that KVM/ARM achieves comparable performance overhead in most cases, and lower performance overhead for two important applications, Apache and MySQL, on ARMv7 multicore platforms. We also show that KVM/ARM provides power efficiency benefits over Linux KVM x86.

2.1 ARM Virtualization Extensions

Because the ARM architecture is not classically virtualizable (See Chapter 1), ARM introduced the Virtualization Extensions (VE) as an optional extension to the ARMv7 [11] and ARMv8 [13] architectures. ARM CPUs such as the Cortex-A15 [10] include this hardware support for virtualization, and all ARMv8 CPUs targeting the server and networking markets include the virtualization extensions. I present a brief overview of the ARM virtualization extensions.

2.1.1 CPU Virtualization

To run VMs, the privileged CPU mode must be virtualized to maintain isolation and for the hypervisor to remain in control of the physical hardware. If VMs were allowed full control of the underlying hardware, they could prevent other tasks and VMs on the system from running, turn off the CPU, and even compromise the integrity of other parts of the system. As explained in Chapter 1, virtualizable architectures can virtualize the privileged CPU mode by running all of the VM, including the guest kernel, in the non-privileged user mode, and trapping each sensitive operation to the hypervisor and emulate the operation in software. To support CPU virtualization on a non-virtualizable architecture such as ARM, one option would be to change the architecture to trap on all sensitive operations, making the architecture virtualizable. However, this approach does not necessarily provide good performance and requires that the hypervisor correctly and efficiently implements emulation of all sensitive operations. If instead the hardware could support executing the sensitive operations directly in the VM by somehow changing these operations to operate on virtual CPU state instead of physical CPU state, then isolation can be achieved without trapping too frequently to the hypervisor. This is the approach taken by both ARM and x86 hardware virtualization support for many operations. For example, when running a VM using the ARM virtualization

extensions, when the VM disables interrupts, this doesn't actually trap to the hypervisor, but instead disables *virtual interrupts*, as will be explained in further detail in Section 2.1.3. ARM implements these features by introducing a new and more privileged CPU mode to run the hypervisor.

Figures 2.1 and 2.2 show the ARM CPU modes and privilege levels, including the optional Security Extensions (TrustZone) and a new privilege level and CPU mode, PL2 (Hyp) for ARMv7 and EL2 for ARMv8. ARM CPUs that implement the security extensions split execution into two worlds, secure and non-secure. A special mode, monitor mode, is provided to switch between the secure and non-secure worlds. Although ARM CPUs implementing the security extensions always power up in the secure world, firmware typically transitions to the non-secure world at an early stage. The secure world is used to run Trusted Execution Environments (TEEs) to support use cases such as digital rights management or authentication. TrustZone may appear useful for virtualization by using the secure world for hypervisor execution, but this does not easily work because the secure world cannot control the non-secure world's access to normal non-secure RAM and can only configure traps from the non-secure to secure world on a few select operations. There are no means to trap general OS operations executed in the non-secure world to the secure world when, for example, an OS configures virtual memory. Any software running at the highest non-secure privilege level therefore effectively has access to all non-secure physical memory, making it impossible to isolate multiple VMs running in the non-secure world. [1]

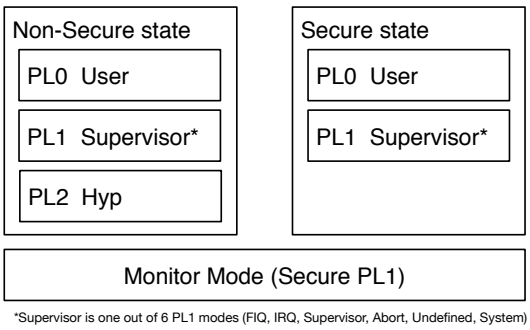

Figure 2.1: ARMv7 Processor Modes

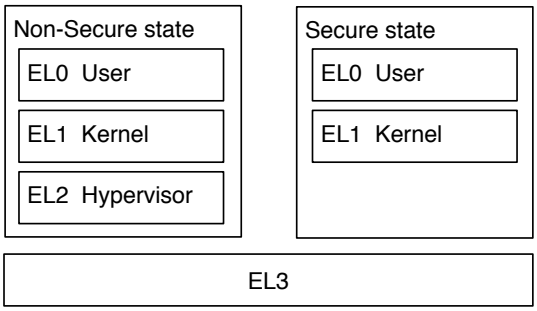

Figure 2.2: ARMv8 Processor Modes

The differences between the ARMv7 and ARMv8 CPU protection mechanisms don't impact

[1] It is potentially possible to build a hypervisor using the security extensions on systems that have a TrustZone Address Space Controller (TSAC) which can be used to disallow non-secure accesses to regions of physical memory, but there are severe limitations to this approach such as not being able to generally virtualize physical memory, but only partition it statically, and it is my opinion that any such solution will perform poorly for general purpose virtualization.

hypervisor designs, but I highlight the important changes here for completeness. The differences relate mostly around the interaction between the non-secure and secure world and to nomenclature. On ARMv7, the privilege level when running in monitor mode is secure PL1, and monitor mode is therefore no more privileged than the PL1 kernel modes. On ARMv8, EL3 is a more privileged mode than secure EL1. For example, ARMv8 defines several system registers that can only be accesses from EL3 and not from secure EL1. This particular difference influences the implementation of how firmware configures the non-secure world, but does not affect the design of hypervisors which are limited to the non-secure state. ARMv7 defines more than one PL1 mode; there are actually six PL1 modes, Supervisor, Abort, Undefined, FIQ, IRQ, and System. These modes share the same privilege level, but have a number of dedicated registers, such as their own stack pointers. System mode is special, because it shares the same registers as user mode, but runs at PL1. ARMv8 does away with multiple modes per privilege level and reduces the complexity to a single concept, the *exception levels*. ARMv8 therefore has three non-secure CPU modes, EL0, EL1, and EL2. When the virtualization extensions were first introduced in ARMv7, the hypervisor CPU mode was named Hyp mode, but when ARMv8 was later introduced and only had a concept of exception levels, the equivalent exception level was called EL2. I therefore use the terms Hyp mode and EL2 interchangeably in this dissertation to refer to the hypervisor execution mode of the CPU.

Hyp mode was introduced to support virtualization on the ARM architecture, and was designed to be as simple as possible to run hypervisors. Hyp mode is strictly more privileged than user and kernel modes. Software running in Hyp mode can configure the hardware to trap from kernel and user mode into Hyp mode on a variety of events. For example, software running in Hyp mode can selectively choose if reading the CPU's ID registers [2] should trap to the hypervisor or if they can be read directly by the VM. It is useful to be able to configure these traps depending on the software configuration, because a hypervisor may choose to emulate the exact same CPU as the underlying hardware CPU to the VM, in which case there is no need to introduce additional performance penalties from trapping on ID register reads, but a hypervisor may also choose to only expose a subset of the hardware CPU's features to the VM, in which case that hypervisor will want to emulate the value of the ID registers. As another example, hypervisor software can selectively choose to trap accesses to floating point registers or allow full access to these registers to the VM. This is useful

[2] ID registers are read-only registers that describe the capabilities of the CPU

when multiplexing the single floating point register bank between multiple VMs, because floating point registers are typically large and therefore expensive to save and restore to memory. Allowing a configurable trap on floating point register access allows a hypervisor to only context switch the floating point register on demand while at the same time giving full access to the floating point hardware to the VM once the register state belongs to the VM's execution context. Traps to the hypervisor from the VM are similar to traps from user space to the kernel and change the CPU mode and jump to a predefined location in memory, known as the exception vector.

To run VMs, the hypervisor must configure traps and memory virtualization from Hyp mode, and perform an exception return to user or kernel mode, depending on whether the VM is going to run its user space or kernel. The VM then executes normally in user and kernel mode until some condition is reached that requires intervention of the hypervisor. At this point, the hardware traps into Hyp mode giving control to the hypervisor, which can then manage the hardware and provide the required isolation across VMs. Once the condition is processed by the hypervisor, the hypervisor can perform another exception return back into user or kernel mode and the VM can continue executing. Not all traps and exceptions are handle by Hyp mode. For example, traps caused by system calls or page faults from user mode trap to a VM's kernel mode directly so that they are handled by the guest OS without intervention of the hypervisor. This avoids going to Hyp mode on each system call or page fault, reducing virtualization overhead. At the same time, hardware interrupts can be configured to always trap to Hyp mode so that the hypervisor can remain in control of the hardware.

The ARM architecture allows fine-grained control over which events and instructions trap from user and kernel mode to Hyp mode. All traps into Hyp mode can be disabled and a single non-virtualized kernel can run in kernel mode and have complete control of the system, effectively bypassing Hyp mode. This is for example the configuration used by Linux when not running a hypervisor.

ARM designed the virtualization support around a separate CPU mode distinct from the existing kernel mode, because the architects envisioned a standalone hypervisor underneath a more complex rich OS kernel and because it fits nicely into the existing privilege hierarchy of the architecture [38]. In an effort to simplify chip implementation and reduce the validation space, ARM reduced the number of control registers available in Hyp mode compared to kernel mode. For example, ARMv7

and later kernel modes have two page table base registers, providing an upper and lower virtual address space for the kernel and user space, but Hyp mode only has a single page table base register. ARM also redefined the meaning of the permission bits in the page table entries used by Hyp mode, because they did not envision a hypervisor sharing page tables with software running in user space. For example, Linux runs in kernel mode, but both user mode and kernel mode use the same page tables, and user space access to kernel data and code is prevented by using permission bits in the page tables which prevent user-mode access to kernel memory, but allows the kernel to directly access user space memory, a very common operation for many Linux system calls such as `read` or `write`.

2.1.2 Memory Virtualization

The ARM architecture provides virtual memory support through a single stage of address translation that translates virtual addresses into physical addresses using page tables. While it is possible to reuse the existing support for virtual memory in the context of VMs, it requires trapping on the VM's access to virtual memory control registers and constructing *shadow page tables*. Shadow page tables are known to add performance overhead and significantly increase implementation complexity [1]. It is therefore important to let the VM manage its own virtual memory structures without trapping to the hypervisor and at the same time allow the hypervisor full control of the physical memory resources. Both requirements can be met by implementing a second stage of address translation in hardware under the control of the hypervisor.

The ARM virtualization extensions provide stage 2 translations as a mechanism for the hypervisor running in Hyp mode to completely control all accesses to physical memory. With two stages of translation, the first stage translates Virtual Addresses (VAs) into Guest Physical Addresses (GPAs), and the second stage translates GPAs into Physical Addresses (PAs). The ARM architecture has its own nomenclature which deviates slightly from commonly used terms; ARM uses the term Intermediate Physical Addresses (IPAs) instead of GPAs. Figure 2.3 shows the complete address translation sequence using three levels of paging for the stage 1 translations and four levels of paging for the stage 2 translations. The number levels of each stage of translation can vary depending on size of the VA, GPA, and PA space. Stage 2 translations, when enabled, translate all GPAs from kernel and user mode into PAs using a dedicated set of page tables. Stage 1 and stage 2 translations can

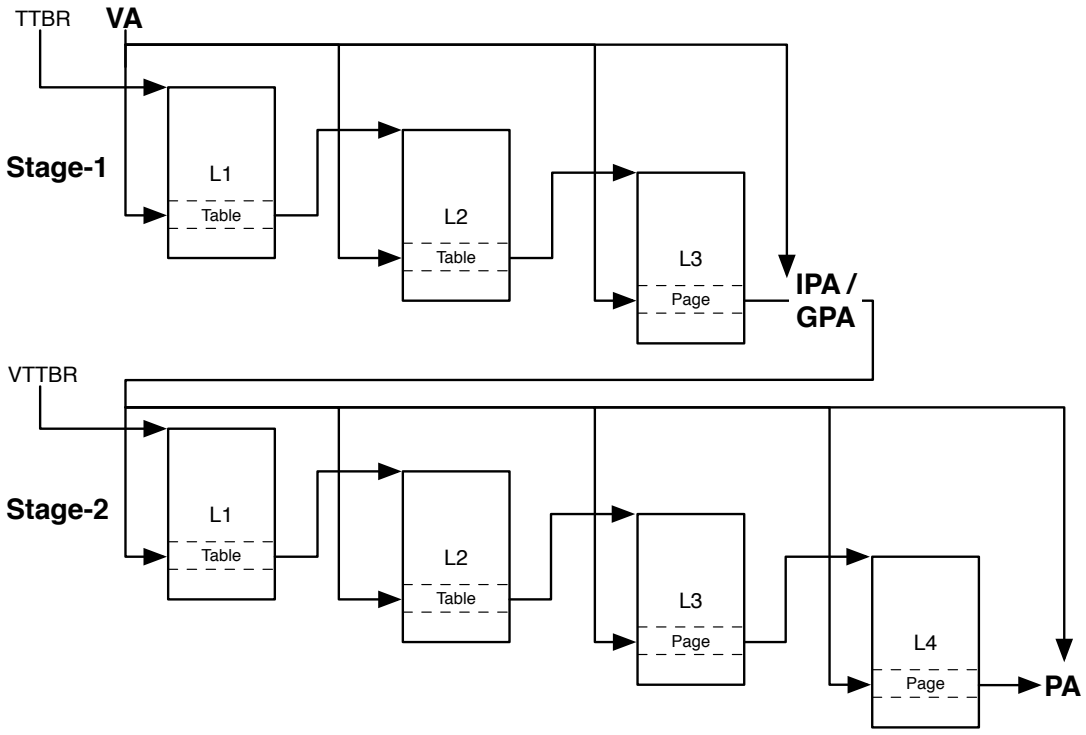

Figure 2.3: Stage 1 and stage 2 Page Table Walk

be independently enabled and disabled. When stage 1 translations are disabled, VAs and GPAs are identical, and similarly, when stage 2 translations are disabled, GPAs and PAs are identical. ARMv7 stage-2 page tables use the LPAE [22] page table format and ARMv8 executing in a 64-bit mode uses the VMSAv8-64 translation table format [13], which is a slightly extended version of LPAE format. The stage 2 page tables use a different format from the stage 1 page tables.

When running a VM, the VM manages its own stage 1 page tables and stage 1 memory control registers without trapping to the hypervisor. The stage 1 page tables translate VAs into GPAs, which are then further translated into PAs using the stage 2 translation system. The stage 2 translation can only be configured from Hyp mode, and can be completely disabled and enabled. The hypervisor therefore manages the GPA to PA translation and non-virtualized operating systems executing in kernel and user mode can manage physical memory directly when stage 2 translation is disabled. Hyp mode uses its own address translation regime, which only supports stage 1 translations using a special page table format (different from the stage 1 page table format used by kernel mode) only used for Hyp mode. Since stage 2 translations are not supported for Hyp mode, the Hyp mode stage 1 page tables translate directly from VAs to PAs, without going through GPAs, because the

hypervisor running in Hyp mode has full control of the physical memory. Stage 2 translations, like stage 1 translations, can be cached in TLB data structures to improve performance. To avoid the need to invalidate TLBs when switching between VMs, ARM supports VMIDs, which are configured on a per-VM basis by the hypervisor and all TLB entries are tagged with the VMID associated with the current stage 2 context.

Both stage 1 and stage 2 page tables allow each page to be marked with a set of memory permissions and attributes. Stage 2 page tables can override the permissions on stage 1 page tables, for example marking otherwise read/write pages as read-only. This is useful for techniques such as page deduplication and swapping. Memory attributes configure if memory accesses are cacheable or non-cacheable, meaning that they bypass caches and access main memory directly. Unfortunately, stage 2 page tables prior to ARMv8.4 do not overwrite the attributes of stage 1 attributes, but instead the *strongest* semantics take precedence. For example, non-cacheable mappings in stage 1 cannot be modified to be cacheable through their stage 2 mappings, because non-cacheable is considered a stronger memory attribute. This design decision limits the ability to emulate certain classes of DMA-capable devices like VGA framebuffers [87].

2.1.3 Interrupt Virtualization

Interrupts are used as an asynchronous notification mechanism from other hardware components to one or more CPUs. For example, I/O devices can notify a CPU of some event that requires the attention of the CPU, and one CPU can signal another CPU to request some action. In the context of virtualization, the hypervisor most typically manages all hardware components capable of generating interrupts, and the hypervisor must therefore still be notified of interrupts from these components even when running a VM. At the same time, the VM will interact with a set of virtual devices and will expect notifications from these virtual devices in the form of interrupts delivered to virtual CPUs. It is therefore important to distinguish between and support *physical interrupts*, generated by hardware components and delivered to physical CPUs, and *virtual interrupts*, generated by virtual devices and delivered to virtual CPUs.

ARM defines the Generic Interrupt Controller (GIC) v2 and v3 architectures [9, 12] which support handling both physical and virtual interrupts. The GIC routes interrupts from devices to CPUs and CPUs query the GIC to discover the source of an interrupt. The GIC is especially important in

multicore configurations, because it is used to generate Inter-Processor Interrupts (IPIs) from one CPU to another. The GIC is split in two parts, the distributor and the CPU interfaces. There is only one distributor in a system, but each CPU has a GIC CPU interface. On GICv2, both the CPU interfaces and the distributor are accessed over a Memory-Mapped interface (MMIO). On GICv3, the CPU interfaces are accessed via system registers and the distributor is still accessed using MMIO. The distributor is used to configure the GIC, for example to configure the CPU affinity of an interrupt, to completely enable or disable interrupts on a system, or to send an IPI to another CPU . The CPU interface is used to acknowledge (ACK) and to deactivate interrupts (End-Of-Interrupt, EOI). For example, when a CPU receives an interrupt, it will read a special register on the GIC CPU interface, which ACKs the interrupt and returns the interrupt number. The interrupt will not be raised to the CPU again before the CPU writes to the EOI register of the CPU interface with the value retrieved from the ACK register, which deactivates the interrupt. [3]

The GIC can signal interrupts to each CPU via two separate signals, IRQs and FIQs. Software can configure the GIC to signal a particular interrupt as an IRQ or FIQ using interrupt groups. (We avoid an intricate discussion of interrupt groups and the relationship to the security extensions here and refer to the architecture manuals for more information.) Each signal, IRQ and FIQ, can be independently configured to trap to either Hyp or kernel mode. Trapping all interrupts to kernel mode and letting OS software running in kernel mode handle them directly is efficient, but does not work in the context of VMs, because the hypervisor loses control over the hardware. Trapping all interrupts to Hyp mode ensures that the hypervisor retains control, but requires emulating virtual interrupts in software to signal events to VMs. This is cumbersome to manage and expensive because each step of interrupt and virtual interrupt processing, such as acknowledging and deactivating, must go through the hypervisor.

The GIC includes hardware virtualization support from v2.0 and later, known as the GIC Virtualization Extensions (GIC VE), which avoids the need to emulate virtual interrupt delivery to the

[3] This is a slightly simplified view, because the GIC supports a concept known as *split priority drop and deactivate*. Once an interrupt has been acknowledged, it enters the active state, and the GIC will not signal that interrupt to the CPU again before the interrupt is deactivated. Normally, a write to the EOI register will deactivate the interrupt and perform a *priority drop*, which means that the GIC will accept other interrupts with the same priority as the most recently acked interrupt. However, with split priority drop and deactivate, a write to the EOI register will only drop the priority, but will not deactivate the interrupt. This is useful to defer some part of handling an interrupt and making sure that a particular interrupt is not seen again until some event passes. For example, this can be used to implement threaded interrupt handling by handing off an acked interrupt to a different thread and be able to take new interrupts of the same priority. With split priority drop and deactivate, software must write to a separate register, the DIR register, after writing to the EOI register.

VM. GIC VE introduces a virtual GIC CPU interface for each CPU and a corresponding hypervisor control interface for each CPU. VMs are configured to see the virtual GIC CPU interface instead of the real GIC CPU interface. Virtual interrupts are generated by writing to special registers, the List Registers (LRs), in the GIC hypervisor control interface, and the virtual GIC CPU interface raises the virtual interrupts directly to a VM's kernel mode. Because the virtual GIC CPU interface includes support for ACK and EOI, these operations no longer need to trap to the hypervisor to be emulated in software, reducing overhead for receiving interrupts on a CPU. For example, emulated virtual devices typically raise virtual interrupts through a software API to the hypervisor, which can leverage GIC VE by writing the virtual interrupt number for the emulated device into the list registers. Upon entering the VM, GIC VE interrupts the VM directly to kernel mode and lets the guest OS acknowledge and deactivate the virtual interrupt without trapping to the hypervisor. Note that the distributor must still be emulated in software and all accesses to the distributor by a VM must still trap to the hypervisor. For example, when a virtual CPU sends a virtual IPI to another virtual CPU, this will cause a trap to the hypervisor, which emulates the distributor access in software and raises the IPI as a virtual IPI to the receiving virtual CPU. If the receiving virtual CPU is executing inside the VM, this involves sending a physical IPI to the CPU that is executing the virtual CPU, which causes the VM virtual CPU to trap to the hypervisor, detect a pending virtual IPI, and program a list register with the virtual IPI on the receiving CPU's GIC hypervisor control interface.

Figure 2.4 shows a simplified overview of GICv2 with the Virtualization Extensions. There are three types of interrupts in the GIC: Private Peripheral Interrupts (PPIs), Shared Peripheral Interrupts (SPIs) and Software-generated Interrupts (SGIs). Devices can signal either PPIs or SPIs to the GIC, which only differ in their allocation of interrupt numbers by having a single interrupt line per SPI, there is a line per PPI per processor. For example, there may be a timer per CPU and the timer interrupts is tied to a PPI, so that it can signal CPUs independently using the same interrupt number. SGIs are generated from within the GIC distributor as a result of MMIO commands from CPUs wishing to send IPIs from one CPU to another. The distributor maintains state and configuration for each interrupt, and forwards pending and enabled interrupts to the CPU interfaces of the receiving CPUs. The CPU interfaces both signal the CPUs using the IRQ and FIQ lines, but they can also be accessed via MMIO from the CPUs to acknowledge and deactivate interrupts, for example. Oper-

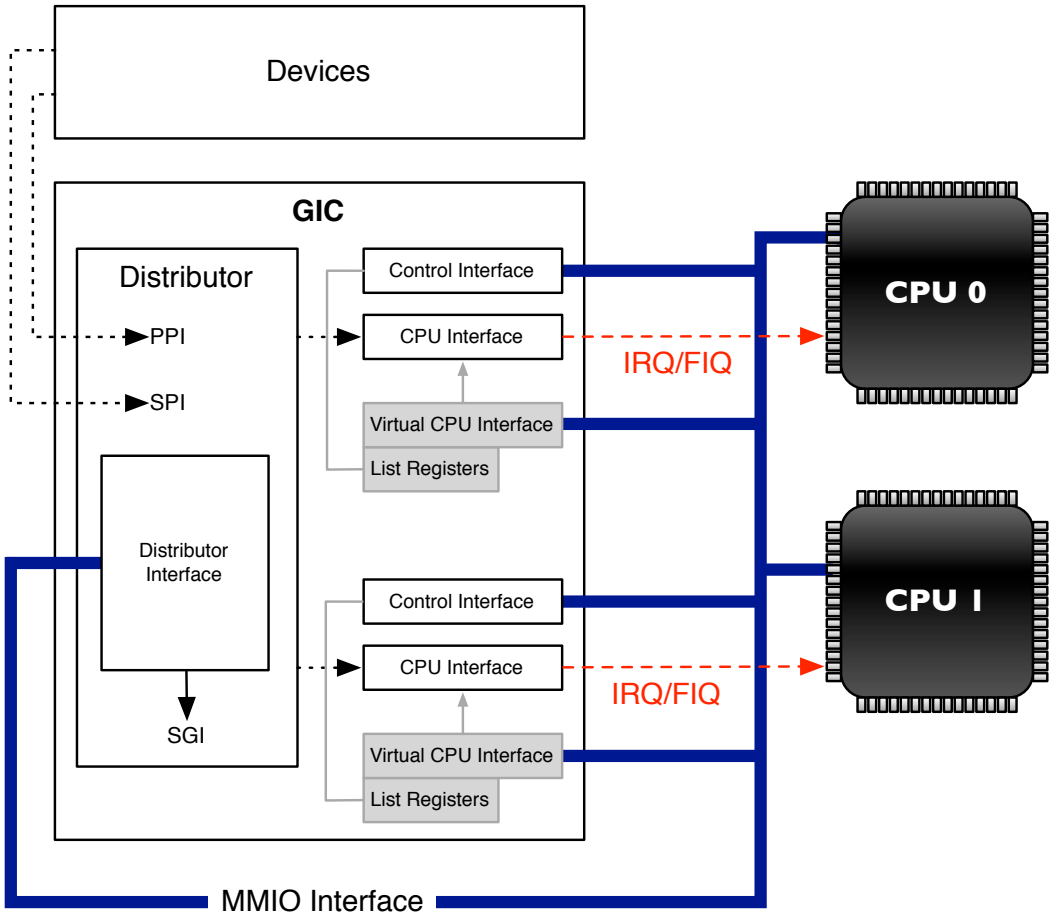

Figure 2.4: ARM Generic Interrupt Controller v2 (GICv2) overview

ating systems running directly on the hardware access the real CPU interfaces, but guest OSes in VMs access the virtual CPU interfaces. Since the GICv2 accesses are all based on MMIO, a hypervisor uses stage 2 page tables to ensure that the VM can only access the virtual CPU interfaces and that VMs cannot access the control interface. The MMIO path shown in Figure 2.4 (indicated by the thick blue line) shows the VM and hypervisor configuration where the CPUs access the hypervisor control interface when executing the hypervisor and access the virtual CPU interface when running the VM. The list registers, controlled via the hypervisor control interface, feed content into the virtual CPU interface, which is observed by the CPUs when running VMs.

Devices can implement interrupt signaling in two ways: Edge-triggered and level-triggered interrupts. Edge-triggered interrupts become pending when a rising edge is observed and a new interrupt is only signaled when the line is lowered and subsequently raised again. Level-triggered

interrupts are pending as long as the level of the line is high, and lose their pending state when the level of the line is low. [4] This means that while the GIC only has to observe changes to edge-triggered inputs, it must constantly sample the level of the line for level-triggered interrupts. To support sampling the line level in the context of VMs, the GIC VE support an additional bit in the list registers, the EOI bit, which causes a trap to the hypervisor when the VM EOIs an interrupt with this bit set. The traps generated from the GIC VE are signaled using a dedicated hardware interrupt, known as the *GIC Maintenance Interrupt*. Hypervisors can set the EOI bit in the list registers for level-triggered interrupts to force an exit from the VM when it has completed a level-triggered interrupt, the hypervisor can then re-sample the virtual line state for the corresponding interrupt, and inject a new interrupt if the level is still held high. Level-triggered interrupts are therefore slower and more complicated to handle than edge-triggered interrupts in the context of VMs.

GIC VE also provides an additional special bit in the list registers, called the HW bit. This functionality is particular to the ARM architecture and fundamental for virtualizing timer hardware, and is therefore worth mentioning in detail. The idea is that virtual interrupts can be tied to physical interrupts such that when a VM deactivates a virtual interrupt, a corresponding physical interrupt can also be deactivated. When the hypervisor receives a physical interrupt, which should be forwarded to the VM as a virtual interrupt, the hypervisor can choose not to deactivate the physical interrupt, but defer this until the VM deactivates the virtual interrupt. This is done by setting the HW bit in the list register used to signal the virtual interrupt and linking the virtual interrupt number to the physical interrupt number, using two separate dedicated fields for this purpose in the list register. (Note that since the VM may present a different platform with different interrupt numbers than the physical host, the virtual and physical interrupt numbers can be different.) This mechanism is useful for hypervisors which do not wish to be interrupted by a device until a VM has completed handling a previous interrupt, which is typically the case when the VM is exposed to any aspect of the underlying hardware, for example in the context of timers or direct device assignment.

[4] Interrupts can actually be implemented electrically differently by either signaling interrupts on a rising or falling edge, or similarly by considering a level-triggered asserted when the line is low or high, and the GIC supports both configurations. This difference has no influence on software design or performance though, and we therefore limit our discussion to interrupts that use rising edges or high levels to assert interrupts.

2.1.4 Timer Virtualization

Systems software needs some mechanism to keep time. Both telling the passing of time and programming timers to generate interrupts after some time has passed are essential in providing OS services to applications. For example, a graphical user interface (GUI) typically programs a timer to blink the cursor when editing text, and OS kernels typically check time spent at various checkpoints to schedule tasks according to a scheduling policy. When running VMs, the hypervisor has similar requirements for both time keeping mechanisms, and the guest OS running inside the VM also need to keep time. However, the definition of time can depend on the underlying hardware and can range from telling wall-clock time to CPU cycles or using other constant timers which may stop counting or count at a slower rate depending on the power state of the hardware system. Furthermore, when running VMs, the VM may not be running continuously because the hypervisor may run other tasks instead of the VM, without the VM being notified, and as a consequence, wall clock time may not relate virtual processing time the same way it relates to native processing time.

The ARM Generic Timer Architecture, also known as the *architected timers*, includes support for timer virtualization. The goal is to provide VMs with access to their own timekeeping hardware, so that VMs can tell time and program timers without trapping to the hypervisor, and to allow VMs to tell the difference between some concept of virtual time and physical time [68]. To meet that goal, ARM provides three separate timers; a hypervisor timer, a physical timer, and a virtual timer. The idea is that the hypervisor timer represents the actual passing of time and is owned exclusively by the hypervisor, where both the physical and virtual timers are used by a VM. The physical timer also represents actual passing of time, where the virtual timer can be adjusted by the hypervisor to provide a measure of virtual time, for example based on when the VM has actually been allowed to run.

More specifically, each timer on ARM comprises both a *counter* and a *timer*. The counter is simply a device that counts up, and the timer is a small logical circuit, which can be programmed to generate an interrupt when the counter reaches a programmable value. The hypervisor timer is only accessible from Hyp mode, and any attempt to program it from kernel or user mode generates an exception to kernel mode. The hypervisor can configure if accesses to the physical timer trap when executed in the VM or if the physical timer is accessible by the VM. The virtual timer is

always accessible in the VM. The hypervisor can program a special virtual counter offset register, only accessible in Hyp mode, which changes the counter value of the virtual timer relative to the physical timer. A hypervisor can use this, for example, to subtract time from the virtual counter when the VM has not been running, and the VM can use this information to adjust its scheduling policy or report *stolen time*, time when the virtual CPU wasn't actually running, to the user.

While the virtualization support for timers allows a VM to directly program a timer without trapping, the hypervisor is still involved in signaling interrupts to a VM when a timer fires. When timers fire, they generate hardware interrupts. Such hardware interrupts are no different from other hardware interrupts on the system, regardless of the timer that signals them, and therefore follow the configuration of the system to either be delivered directly to kernel mode, when executing a native OS, or trap to the hypervisor when executing a VM. When the hypervisor handles an interrupt from a timer device which is assigned to a VM, for example the virtual timer, it will typically program a virtual interrupt using the GIC VE to signal to the VM that the timer has expired. However, since the timer output signal is level-triggered, the timer output will remain asserted until the VM, which has direct access to the timer hardware, cancels the timer or reprograms it to fire in the future. If the hypervisor deactivates the interrupt on behalf of the VM, the GIC will re-sample the output line from the timer, and since it is still asserted, it will immediate signal the new pending interrupt to the CPU, and the CPU will not make any forward progress, because it will be busy handling a continuous stream of interrupts. The ARM architecture is specifically designed to integrate the functionality of the timers with the GIC, by leaving the physical interrupt from a timer assigned to a VM active, which prevents further signaling of interrupts, and inject a virtual interrupt for the timer to VM, which is tied to the underlying physical interrupt by setting the HW bit in the list register. In that way, when the VM runs and observes the pending virtual timer, it will run the Interrupt Service Routine (ISR) for the timer, cancel the timer which lowers the timer output signal, and will finally deactivate the interrupt, for example by writing to the EOI register on the virtual CPU interface. Deactivating the virtual interrupt deactivates both the physical and virtual interrupt because the HW bit is set, and when the timer fires again later, it will trap to the hypervisor.

Each CPU has its own private set of timers, and it is the hypervisor's responsibility to properly migrate and synchronize timers when migrating virtual CPUs across physical CPUs.

2.1.5 Comparison of ARM VE with Intel VMX

Virtualization Support	ARM VE	Intel VMX
CPU	Separate CPU Mode	Duplicated CPU modes (root vs. non-root operation)
VM/hypervisor transition	Trap (software save/restore)	VMX Transition (hardware save/restore with VMCS)
Memory	Stage 2 page tables	EPT (2nd+ gen.)
Direct Timer Access	Physical/Virtual Timers	No equivalent support
Interrupts	VGIC	APICv
Direct virtual interrupt injection	GICv4	Posted Interrupts

Table 2.1: Comparison of ARM VE with Intel VMX

There are a number of similarities and differences between the ARM virtualization extensions and hardware virtualization support for x86 from Intel and AMD. Intel VMX [50] and AMD-V [2] are very similar, so we limit our comparison to ARM VE and Intel VMX. The differences are summarized in Table 2.1. ARM supports virtualization through a separate CPU mode, Hyp mode, which is a separate and strictly more privileged CPU mode than previous user and kernel modes. In contrast, Intel has root and non-root mode [50], which are orthogonal to the CPU protection modes. While sensitive operations on ARM trap to Hyp mode, sensitive operations can trap from non-root mode to root mode while staying in the same protection level on Intel. A crucial difference between the two hardware designs is that Intel's root mode supports the same full range of user and kernel mode functionality as its non-root mode, whereas ARM's Hyp mode is a strictly different CPU mode with its own set of features. For example, software executing in Intel's root mode use the same virtual memory system and same virtual address layout as software executing in non-root mode. On ARM, Hyp mode uses different page tables and has a limited VA space compared to kernel modes. These differences suggest that Intel imagined any software stack, similar to existing OSes, running in both root and non-root mode, where ARM imagined a smaller standalone hypervisor running in root mode.

Both ARM and Intel trap into their respective Hyp and root modes, but Intel provides specific

hardware support for a VM control structure (VMCS) in memory which is used to automatically save and restore the CPU's execution state when switching to and from root mode using the VMX transitions VM Entry and VM Exit. These transitions are defined as single atomic operations in hardware and are used to automatically save and restore all of the CPU's execution state using the VMCS when switching between guest and hypervisor execution contexts. 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. For example, trapping to ARM's Hyp mode on first-generation ARM VE hardware only costs tens of cycles compared to hundreds of cycles to transition from root mode to non-root mode on optimized x86 server hardware. ARM and Intel are quite similar in their support for virtualizing physical memory. Both introduce an additional set of page tables to translate guest to host physical addresses. ARM benefited from hindsight in including stage 2 translation whereas Intel did not include its equivalent Extended Page Table (EPT) support until its second generation virtualization hardware.

ARM's support for virtual timers have no real x86 counterpart. The x86 world of timekeeping consists of a myriad of timekeeping devices available partially due to the history and legacy of the PC platform. Modern x86 platforms typically support an 8250-series PIT for legacy support and a Local APIC (LAPIC) timer. The Intel hardware support for virtualization adds the VMX-Preemption timer which allows hypervisors to program an exit from a VM independently from how other timers are programmed. The VMX-Preemption timer was added to reduce the latency between a timer firing and the hypervisor injecting a virtual timer interrupt. This is achieved because a hypervisor doesn't have to handle an interrupt from the LAPIC timer, but can directly tell from a VM exit that the preemption timer has expired. Contrary to ARM, x86 does not support giving full control of individual timer hardware to VMs. x86 does allow VMs to directly read the Time Stamp Counter (TSC), whereas ARM allows access to the virtual counter. The x86 TSC is typically higher resolution than the ARM counter, because the TSC is driven by the processor's clock where the ARM counter is driven by a dedicated clock signal, although with minimum 50 MHz frequency.

Recent Intel CPUs also include support for APIC virtualization [50]. Similar to ARM's GIC

VE support for virtual interrupts, Intel's APIC virtualization support (APICv) also allows VMs to complete virtual interrupts without trapping to the hypervisor. Also similar to ARM, APICv allows the VM to access interrupt controller registers directly without trapping to the hypervisor. On ARM, the GIC virtual CPU interface supports direct access to all CPU interface registers. APICv provides a backing page in memory for the VM's virtual APIC state. On Intel without APICv, completing virtual interrupts in the VM traps to root mode.

Both ARM and x86 have introduced support for direct delivery of virtual interrupts. Direct delivery means that a CPU executing a VM can observe a new virtual interrupt which is signaled to the VM directly from passthrough devices without ever having to trap from the VM to the hypervisor. Intel has introduced posted interrupts as part of the Intel Directed I/O Architecture [51]. Posted interrupts support directly delivering virtual interrupts from physical devices or from other CPUs. ARM has introduced version 4.0 of the GIC architecture (GICv4) [12], which supports directly delivering virtual Message-Signaled Interrupts (MSIs) from physical devices signaling MSIs. Unfortunately, no hardware with GICv4 is available at the time of writing this dissertation and I therefore focus on GICv2 and GICv3 where it is possible to validate and evaluate virtualization software using them.

2.2 KVM/ARM System Architecture

Instead of reinventing and reimplementing complex core functionality in the hypervisor, and potentially introducing tricky and fatal bugs along the way, KVM/ARM builds on KVM and leverages existing infrastructure in the Linux kernel. While a standalone hypervisor design approach has the potential for better performance and a smaller Trusted Computing Base (TCB), this approach is less practical on ARM. ARM hardware is in many ways much more diverse than x86. Hardware components are often tightly integrated in ARM devices in non-standard ways by different device manufacturers. For example, much ARM hardware lacks features for hardware discovery such as a standard BIOS or a PCI bus, and there is no established mechanism for installing low-level software on a wide variety of ARM platforms. Linux, however, is supported across almost all ARM platforms and by integrating KVM/ARM with Linux, KVM/ARM is automatically available on any device running a recent version of the Linux kernel. This is in contrast to bare metal approaches

such as Xen [26], which must actively support every platform on which they wish to install the Xen hypervisor. For example, for every new SoC that Xen wants to support, the developers must implement SoC support in the core Xen hypervisor to support simply outputting content on a serial console. Booting Xen on an ARM platforms also typically involves manually tweaking boot loaders or firmware, since Xen requires booting two images at the same time, Xen and the Linux Dom0 kernel image. KVM/ARM on the other hand is integrated with Linux and follows existing supported Linux kernel boot mechanisms.

While KVM/ARM benefits from its integration with Linux in terms of portability and hardware support, a key problem we had to address was that the ARM hardware virtualization extensions were designed to support a standalone hypervisor design where the hypervisor is completely separate from any standard kernel functionality, as discussed in Section 2.1. The following sections describe how KVM/ARM's novel design makes it possible to benefit from integration with an existing kernel and at the same time take advantage of the hardware virtualization features.

2.2.1 Split-mode Virtualization

Simply running a hypervisor entirely in ARM's Hyp mode is attractive since it is designed specifically to run hypervisors and to be more privileged than other CPU modes used to run VMs. However, since KVM/ARM leverages existing kernel infrastructure such as the scheduler and device drivers, running KVM/ARM in Hyp mode requires running the entire Linux kernel in Hyp mode. This is problematic for at least two reasons:

First, low-level architecture dependent code in Linux is written to work in kernel mode, and would not run unmodified in Hyp mode, because Hyp mode is a completely different CPU mode from normal kernel mode. For example, the kernel's exception entry path, process switching logic, and many other kernel subsystems access kernel mode registers, and would have to be modified to work in Hyp mode. Modifying a full OS kernel such as the Linux kernel to run in Hyp mode is invasive and controversial, and, more importantly, to preserve compatibility with hardware without Hyp mode and to run Linux as a guest OS, low-level code would have to be written to work in both Hyp and kernel mode, potentially resulting in slow and convoluted code paths. As a simple example, a page fault handler needs to obtain the virtual address causing the page fault. In Hyp mode this address is stored in a different register than in kernel mode.

Second, running the entire kernel in Hyp mode would adversely affect native performance. For example, Hyp mode has its own separate address space. Whereas kernel mode uses two page table base registers to provide a split between user and kernel virtual address space, Hyp mode uses a single page table base register and therefore cannot easily have direct access to the user address space. Supporting user access from the kernel would involve either using the same page tables from different translation regimes, which relaxes isolation guarantees and increases the TLB invalidation frequency, or modifying frequently used functions to access user memory to walk process page tables in software and explicitly map user space data into the kernel address space and subsequently perform necessary page table teardown and TLB maintenance operations. Both options are likely to result in poor native performance on ARM and would be difficult to integrate in the mainline Linux kernel. Chapter 4 explores the challenges of running Linux in the ARMv8 equivalent of Hyp mode, EL2, in more detail.

These problems with running Linux in the hypervisor CPU mode do not occur for x86 hardware virtualization. x86 root operation is orthogonal to its CPU privilege modes. The entire Linux kernel can run in root operation because the same set of CPU modes are available in both root and non-root operations.

KVM/ARM introduces split-mode virtualization, a new approach to hypervisor design that splits the core hypervisor so that it runs across different privileged CPU modes to take advantage of the specific benefits and functionality offered by each CPU mode. KVM/ARM uses split-mode virtualization to leverage the ARM hardware virtualization support enabled by Hyp mode, while at the same time leveraging existing Linux kernel services running in kernel mode. Split-mode virtualization allows KVM/ARM to be integrated with the Linux kernel without intrusive modifications to the existing code base.

This is done by splitting the hypervisor into two components, the lowvisor and the highvisor, as shown in Figure 2.5. The lowvisor is designed to take advantage of the hardware virtualization support available in Hyp mode to provide three key functions. First, the lowvisor sets up the correct execution context by appropriate configuration of the hardware, and enforces protection and isolation between different execution contexts. The lowvisor directly interacts with hardware protection features and is therefore highly critical and the code base is kept to an absolute minimum. Second, the lowvisor switches from a VM execution context to the host execution context and vice-versa.

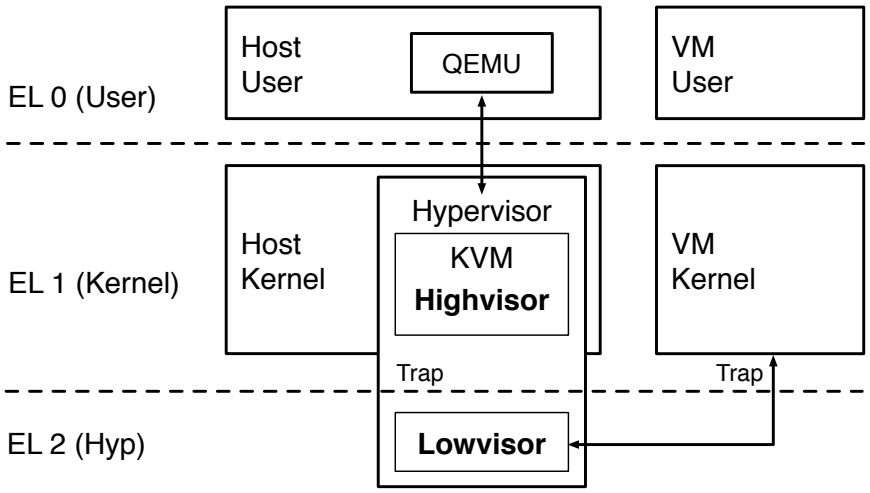

Figure 2.5: KVM/ARM System Architecture

The host execution context is used to run the hypervisor and the host Linux kernel. Transitioning between the host execution context and the VM execution context is often referred to as a *world switch*, because the entire kernel and user mode configurations are changed, giving the impression of moving between worlds. Since the lowvisor is the only component that runs in Hyp mode, only it can be responsible for the hardware reconfiguration necessary to perform a world switch. Third, the lowvisor provides a virtualization trap handler, which handles interrupts and exceptions that must trap to the hypervisor. The lowvisor performs only the minimal amount of processing required and defers the bulk of the work to be done to the highvisor after a world switch to the highvisor is complete.

The highvisor runs in kernel mode as part of the host Linux kernel. It can therefore directly leverage existing Linux functionality such as the scheduler, and can make use of standard kernel software data structures and mechanisms to implement its functionality, such as locking mechanisms and memory allocation functions. This makes higher-level functionality easier to implement in the highvisor. For example, while the lowvisor provides a low-level trap handler and the low-level mechanism to switch from one world to another, the highvisor handles stage 2 page faults from the VM and performs instruction emulation. Note that parts of the VM run in kernel mode, just like the highvisor, but with stage 2 translation and trapping to Hyp mode enabled.

Because the hypervisor is split across kernel mode and Hyp mode, switching between a VM and the highvisor involves multiple transitions between CPU modes. A trap to the highvisor while

running the VM will first trap to the lowvisor running in Hyp mode. The lowvisor will then perform an exception return to kernel mode to run the highvisor after having reconfigured the protection settings for kernel mode. Similarly, going from the highvisor to a VM requires trapping from kernel mode to Hyp mode, and then switching to the VM. As a result, split-mode virtualization incurs a *double trap cost* in switching to and from the highvisor. [5] The only way to enter Hyp mode from kernel mode on ARM is to cause a trap, for example by issuing a hypercall (`HVC`) instruction, because Hyp mode is a more privileged mode and therefore the CPU protection mechanism is designed specifically that way. However, as shown in Section 2.4, this extra trap is not a significant performance cost on ARM.

When going from the highvisor to the VM, via the lowvisor, the highvisor and lowvisor have to communicate with each other. For example, the highvisor must specify the state of the VCPU which is about to run. KVM/ARM uses a memory mapped interface to share data between the highvisor and lowvisor. The lowvisor, running in Hyp mode, uses a separate virtual address space, configured using control registers by the lowvisor itself. One simplistic approach would be to reuse the host kernel's page tables and also use them in Hyp mode to make the address spaces identical. This unfortunately does not work, because Hyp mode uses a different page table format from kernel mode. However, the page tables used by Hyp mode, are just pages in memory, and the highvisor has full access to all memory on the system, and can therefore freely configure the page tables used by Hyp mode. This is a subtle but important insight; even though Hyp mode manages the virtual memory control registers, as long as the highvisor knows the address of the page tables, the highvisor can configure the virtual memory mapping of Hyp mode. The only restriction is, that the highvisor must ask Hyp mode to perform Hyp-specific TLB invalidations if changing any already established mappings, because the Hyp mode TLB invalidation instructions are only available in Hyp mode. KVM/ARM allocates a full set of page tables to map the full Hyp VA space during boot, and passes the pointer to the page tables on to the lowvisor during the initial configuration of the lowvisor. Now, the highvisor can allocate a memory page, map it in both the highvisor's address space and in the lowvisor's address space, and translate pointers from the highvisor's address space to the lowvisor's address space, and pass on the pointers to the lowvisor, which can then access data

[5] We refer to this cost as the *double trap cost* even though it is technically a question of a trap and an exception return, but, since an exception return is an instruction synchronizing event with similar performance characteristics as a trap, the term is only mildly misleading.

in the shared pages. This technique is also used by the highvisor to map code segments during the initial configuration of the lowvisor.

A remaining question is how to establish an efficient runtime for the lowvisor. The initial KVM/ARM implementation only mapped a single page of hand-written assembly code to implement the lowvisor functionality, and mapped shared data structures per VM in Hyp mode every time a new VM was created. As it turned out, running C code in the lowvisor only required linking the lowvisor code in a separate section so that it can be mapped in Hyp mode, map the read-only segments in Hyp mode, and map a page per CPU to use as a stack in Hyp mode. Using C code made it easier to maintain the lowvisor implementation and add logic for things such as support for debugging the VM. Note that even when the lowvisor is implemented in C, it cannot freely access all kernel data structures or call into kernel functions in the host OS, because it still runs in a separate address space and only the lowvisor runtime and the explicitly shared data structures are available.

2.2.2 CPU Virtualization

To virtualize the CPU, KVM/ARM must present an interface to the VM which is essentially identical to the underlying real hardware CPU (the equivalence property), while ensuring that the hypervisor remains in control of the hardware (the isolation property). The CPU state is contained in registers, both general purpose registers and control registers, and KVM/ARM must either trap and emulate access to these registers or context switch them. Since Hyp mode is completely decoupled from kernel and user mode, using its own configuration register state, all kernel and user mode state can be context switched by the lowvisor when transitioning between the VM and the host. Some registers, however, which are not often used, can be configured to trap when accessed, and context switched at a later time to improve performance.

Even though Hyp mode is separate from the VM, the lowvisor still has to ensure that it will eventually regain control of the CPU, for example because the hardware traps when there is a timer interrupt. The lowvisor accomplishes this by always scheduling a timer in the future and configuring the CPU to trap to Hyp mode when receiving a physical interrupt. That way, even if the VM runs an endless loop, for example, the hardware will always eventually trap to the lowvisor, which can preempt the VM and switch back to the highvisor.

Tables 2.2 and 2.3 show the CPU state visible to software running in kernel and user mode

Number of registers	State
28	General Purpose (GP) Registers [1]
6	Program Status Registers [2]
23	CP15 Control Registers
7	VGIC v2 Control Registers
4	VGIC v2 List Registers [3]
2	Arch. Timer Control Registers
32	64-bit VFP registers (VFPv3-D32)
3	32-bit VFP Control Registers
105	Total number of registers

[1] Including banked FIQ registers, SPs and LRs
[2] Banked SPSRs and the CPSR
[3] 4 LRs on a Cortex-A15

Table 2.2: ARMv7 CPU State

Number of registers	State
33	General Purpose (GP) Registers [1]
2	Program Status Registers [2]
37	General System Registers
7	VGIC v2 Control Registers
4	VGIC v2 List Registers [3]
2	Arch. Timer Control Registers
64	64-bit VFP registers
3	32-bit VFP Control Registers
151	Total number of registers

[1] Including LR and SP_EL0 and SP_EL1
[2] SPSR_EL2 and SPSR_EL1
[3] 4 LRs on a Cortex-A57

Table 2.3: ARMv8 CPU state (AArch64)

on both the host and VM which must be context switched when switching between a VM and the host. The number of registers specifies how many register of each type there are. Each of those registers may both be read and written, both when transitioning from the VM to the host and vice versa. For example, consider any of the general purpose registers. KVM/ARM must first read the VM's GP register state and store it to memory, then load the host's GP register state from memory to the register, and then return to the highvisor. Upon returning to the VM, KVM/ARM must again read the host's GP register state and store it memory, and finally load the VM's GP register state from memory to the register, and the return to the VM. Tables 2.2 and 2.3 count each register once, regardless of how many times each register is accessed. Further, the tables only include the minimal set of registers to run workloads under normal operations in the VM, but do not include

more advanced features such as monitoring performance in VMs using the Performance Monitoring Unit (PMU) or debugging the VM.

The VFP floating point registers are costly to save and restore and they are not always used by the VM. Therefore, KVM/ARM performs *lazy context switching* of these registers which means that the registers are initially configured to trap to the lowvisor when accessed, and the lowvisor does not save and restore them on every transition between the VM and the hypervisor. When the VM eventually tries to access the floating point register, it traps to the lowvisor, which then context switches all the VFP registers state between the host and the VM, disables traps on the VFP registers, and returns to the VM which can now access the registers directly.

Trap	**Description**
WFI	Wait-for-interrupts
WFE	Wait-for-event
SMC	Secure Monitor Call
PMU Registers	Performance Monitor Unit Registers
Debug Registers	Debugging controls, breakpoints, and watchpoints
Stage 1 MMU control registers	Only until guest enables the stage 1 MMU

Table 2.4: Instructions and registers that trap from the VM to the hypervisor

KVM/ARM performs trap on certain sensitive instructions and register accesses as shown in Table 2.4. KVM/ARM traps if a VM executes the WFI instruction, which causes the CPU to power down, because such an operation should only be performed by the hypervisor to maintain control of the hardware. KVM/ARM traps on WFE instructions when physical CPUs are oversubscribed, because the WFE instruction is typically used in spinlock loops, and the guest issuing the instruction is a good indication that a lock in the VM is contended and will not be released before the holding VCPU is allowed to run. With oversubscribed CPUs, where more VCPUs are running than there are physical CPUs, KVM/ARM handles traps from WFE by scheduling another VCPU. SMC instructions are trapped because the VM should not be allowed to interact with any secure runtime on the system, but such accesses should instead be emulated by the hypervisor. Debug and PMU registers are trapped because the performance monitoring unit and the debug hardware belong to the host OS and do not have support for virtualization and the hypervisor therefore instead traps and emulates guest accesses to these resources. Finally, when initially booting a VM, KVM/ARM traps access to the VM's stage 1 MMU control registers, and also context switches these registers,

because the hypervisor must be able to detect when the VM enables the stage 1 MMU and perform required cache maintenance to ensure a coherent memory view from the guest. Once the guest OS has enabled the stage 1 MMU, the traps are disabled and the VM can directly access the MMU control registers.

Note that both the VM and the hypervisor execute code in kernel and user mode, but these modes are configured differently depending on the execution context. For example, the lowvisor configures the CPU to trap on WFI instructions when switching to a VM and disables those traps again when switching back to the hypervisor, letting the host OS power down CPUs when needed. There are different configuration settings of the hardware for running the VM and the host, both using the same kernel and user CPU modes. The lowvisor, running in Hyp mode, configures these settings in the following way when running a VM:

1. Store all host GP registers on the Hyp stack.
2. Configure the VGIC for the VM.
3. Configure the timers for the VM.
4. Save all host-specific configuration registers onto the Hyp stack.
5. Load the VM's configuration registers onto the hardware, which can be done without affecting current execution, because Hyp mode uses its own configuration registers, separate from the host state.
6. Configure Hyp mode to trap floating-point operations for lazy context switching, trap interrupts, trap CPU halt instructions (WFI/WFE), trap SMC instructions, trap specific configuration register accesses, and trap debug register accesses.
7. Write VM-specific IDs into shadow ID registers.
8. Set the stage 2 page table base register (VTTBR) and enable Stage-2 address translation.
9. Restore all guest GP registers.
10. Perform an exception return into either user or kernel mode.

The CPU will stay in the VM world until an event occurs, which triggers a trap into Hyp mode. Traps can be caused by any of the traps listed in Table 2.4 and shown in step 6 above, a stage 2 page fault, or a hardware interrupt. Since the event requires services from the highvisor, either to emulate the expected hardware behavior for the VM or to service a device interrupt, KVM/ARM must switch back into the highvisor and the host. Switching back to the host from the VM performs the following actions:

1. Store all VM GP registers.
2. Disable stage 2 translation to give the host full control of the system and all memory.
3. Configure Hyp mode to not trap any register access, instructions, or interrupts.
4. Save all VM-specific configuration registers.
5. Load the host's configuration registers onto the hardware.
6. Configure the timers for the host.
7. Save VM-specific VGIC state.
8. Restore all host GP registers.
9. Perform an exception return into kernel mode.

2.2.3 Memory Virtualization

KVM/ARM provides memory virtualization by enabling stage 2 translation for all memory accesses when running in a VM. Stage 2 translation can only be configured in Hyp mode, and its use is completely transparent to the VM. The highvisor manages the stage 2 translation page tables to only allow access to memory specifically allocated for a VM; other accesses will cause stage 2 page faults which trap to the hypervisor. This mechanism ensures that a VM cannot access memory belonging to the hypervisor or other VMs, including any sensitive data. Stage 2 translation is disabled when running in the highvisor and lowvisor because the highvisor has full control of the complete system and directly manages physical memory using its own stage 1 virtual to physical mappings. When the hypervisor performs a world switch to a VM, it enables stage 2 translation and

configures the stage 2 page table base register to point to the stage 2 page tables configured for the particular VM by the highvisor. Although both the highvisor and VMs share the same CPU modes, stage 2 translations ensure that the highvisor is protected from any access by the VMs.

KVM/ARM uses split-mode virtualization to leverage existing kernel memory allocation, page reference counting, and page table manipulation code. KVM/ARM handles stage 2 page faults by considering the GPA of the fault, and if that address belongs to normal memory in the VM memory map, KVM/ARM allocates a page for the VM by simply calling an existing Linux kernel memory allocation function, and maps the allocated page to the VM in the stage 2 page tables. Further, Linux also brings additional benefits for free. Kernel Same-Page Merging (KSM) finds identical pages in multiple VMs and merges them into a single page to save memory. The pages are marked read-only, and split into multiple pages if a VM writes to the page at a later time. Transparent Huge Pages (THP) improve performance by merging consecutive physical pages into a single bigger page. Larger pages only occupy a single entry in the TLB instead of multiple entries, covering the same virtual address space, and therefore reduces TLB pressure. On ARM, larger pages also require one less level of page walks on a TLB miss, which also improves performance. In comparison, a bare metal hypervisor would be forced to either statically allocate memory to VMs or write an entire new memory allocation subsystem.

KVM/ARM manages memory for VMs by leveraging existing Linux concepts like processes and threads. Each Linux process has its own virtual address space, which is shared between all the threads in that process, which is analogous to a machine which has a single physical memory resource and multiple CPUs which share the same view of memory. KVM/ARM represents each VM as a process and each VCPU as a thread in that process, sharing the same memory management structure. KVM/ARM relies on user space to manage the physical memory map of a VM. User space allocates a virtual memory region using standard memory allocation functions, such as `malloc` and `mmap` for each contiguous region of guest physical memory. User space then calls an ioctl into KVM telling KVM that a particular virtual memory region should be considered a guest physical memory region at a specified offset. For example, for a VM with 2GB of contiguous guest physical memory located at 0x40000000, user space would allocate 2GB of virtual address space and would tell KVM that the allocated region, at some virtual address, corresponds to a 2GB physical memory region at 0x40000000. When a VM begins to consume new memory, it will cause stage

2 page faults when the MMU translates the GPA to a PA. The KVM/ARM lowvisor will capture this GPA from Hyp control registers and pass it up to the highvisor via the shared memory interface. The highvisor can then look up the physical memory region registered by user space, and calculate the offset from the base of the guest physical memory region, 0x40000000 in our example above. Adding the offset to the address of the memory region in the VM process's virtual address space results in a normal Linux process VA, and KVM allocates a physical page by calling standard Linux functionality e.g. `get_user_pages`.

KVM/ARM can very easily leverage more advanced memory management techniques from Linux. For example, swapping pages to disk, an important feature in server and desktop deployments, becomes almost trivially supported, because Linux simply follows its normal routines under memory pressure to swap pages to disk. A key advantage to KVM over other hosted hypervisors is the ability to change other parts of the kernel to provide interfaces needed for hypervisor functionality. For example, VMware workstation running on Windows has to rely on existing APIs from the Windows kernel, where KVM developers have simply modified Linux to cater to the needs of KVM. One such feature is the Linux memory management notifiers [28], which were a somewhat invasive addition to the Linux memory management subsystem introduced specifically for better VM support. KVM/ARM uses these notifiers to registers with the Linux memory management subsystem to be notified of pages belonging to a VM process being swapped out. KVM/ARM then simply has to scan its stage 2 page tables for a mapping to that page, remove the mapping from the stage 2 page table and invalidate the TLB. Transparent Huge Pages (THP) work in a similar way, as long as user space allocates a virtual memory region aligned at the size of a huge page (typically 2MB on ARM using 4K pages), KVM/ARM asks Linux when allocating a page, if the page belongs to a huge page, and in that case maps 2MB of physically contiguous GPA space to a PA using a single stage 2 page table entry.

2.2.4 I/O Virtualization

At a high level, KVM/ARM's I/O model is very similar to the KVM I/O model for other architectures, but the underlying implementation details vary slightly. KVM roughly supports three different kinds of virtualized I/O: (1) Emulated I/O, (2) Paravirtualized I/O, (3) Direct Device Assignment.

Emulated I/O is supported in user space of the host OS, where a user space KVM driver, typ-

ically QEMU, emulates a real device. With the exception of direct device assignment, which is typically only used for a small subset of devices such as network adapters, KVM/ARM never allows VMs direct physical access to devices. Therefore, when a VM tries to communicate with a device, it will trap to the hypervisor. On ARM, all I/O operations are based on memory-mapped I/O (MMIO), and access to devices is controlled via the stage 2 page tables. This is somewhat different from x86, which uses x86-specific hardware instructions such as `inl` and `outl` for port I/O operations in addition to MMIO. When a VM accesses a device, it will therefore generate a stage 2 page fault, which traps to the lowvisor in Hyp mode. The lowvisor can read back system registers, which contain information about the instruction that caused the trap and are only available in Hyp mode. These registers contain important information such as the address that caused the fault, the size of the access, if it was a read or write operation, and which register was used as the source or destination for the access. KVM forwards this information, together with the register payload, if the trap was a write, back up to the host user space application which routes the request to the emulated device based on the faulted GPA, and the user space KVM driver emulates the access. Since this path of trapping on a stage 2 fault, changing context back to the host OS, and forwarding requests from the host kernel to host user space is much more complicated and much slower than interacting with a real hardware device, the technique is mostly used for devices where bandwidth and latency is not a concern. For example, this technique is typically used for emulated serial ports (UARTs), real-time clocks, and low-bandwidth USB devices like mice and keyboards. On x86, this technique is also used to emulate IDE and floppy disks to support OS installers and early VM boot code.

Paravirtualized I/O is mainly used to improve the performance of block storage, and network connections. Instead of emulating a real hardware interface, which may require many traps between the VM and the host OS user space, paravirtualized I/O uses an interface specifically designed and optimized to communicate between a VM and the hypervisor backend. A popular communication protocol developed for this purpose is Virtio [70, 71, 79] which describes how a series of devices can communicate efficiently across the VM and hypervisor boundary. For example, a Virtio device can either be emulated as a directly mapped MMIO device on an emulated ARM SoC, or it can be emulated as a device plugged into a virtual emulated PCI bus. Both options are supported in recent versions of KVM/ARM and Linux on ARM (Note that earlier versions of Linux on ARM only supported directly mapped devices, because PCI was not yet standardized and supported on ARM.)

Virtio coalesces data transfers by establishing shared memory regions between the VM and the host operating system, such that either side can place large amounts of data into memory and signal the other end as rarely as possible. For example, if the VM wishes to send data over the network, it can put several network packages of data into memory, and finally perform a single write to a register in the Virtio device, which traps to the hypervisor, resulting in fewer transitions between the VM and the hypervisor. A further improvement to paravirtualized I/O known as `VHOST` moves much of the device emulation from host user space into the host kernel, such that KVM does not need to go to user space to perform I/O on behalf of the VM. This works by communicating with other threads inside the kernel via file descriptors used to map memory regions and signal events and interrupts. KVM/ARM fully supports VHOST with Virtio over PCI devices and reuses much of the existing KVM and Linux infrastructure.

Direct device assignment works by assigning a dedicated hardware device to the VM, letting the VM directly access the physical device. KVM/ARM again supports this configuration through its integration with Linux. Another Linux subsystem, `vfio`, makes device resources such as memory regions, DMA regions, and interrupts available to user space processes. VFIO is carefully designed to on operate on devices that can be safely assigned to VMs without violating the isolation requirement for VMs, for example by using safety measures such as IOMMUs to isolate DMA. KVM again benefits from representing VMs as processes, and can easily detect VFIO-assigned devices in a process and establish mappings to those devices inside a VM by setting up the required stage 2 page table mappings. However, devices capable of performing DMA do not use the CPU's stage 2 page tables, but can potentially access all of the system memory. If a VM is allowed to freely program a device to DMA to any memory, the VM can use directly assigned devices to break isolation and gain access to sensitive information. To avoid this problem, direct device assignment is only supported on systems with an IOMMU that can translate all DMA requests from devices assigned to VMs. KVM leverages VFIO to ensure that the IOMMU limits memory transactions for an assigned device to memory designated as DMA regions in the KVM VM process. Any other access causes an IOMMU fault, which will trap to the hypervisor.

A more detailed discussion of I/O virtualization performance on ARM and x86 is provided in Chapter 3 and Chapter 4.

2.2.5 Interrupt Virtualization

KVM/ARM leverages its tight integration with Linux to reuse existing device drivers and related functionality, including handling interrupts. When running in a VM, KVM/ARM configures the CPU to trap all hardware interrupts to Hyp mode. When an interrupt occurs, the CPU traps from the VM to the lowvisor in Hyp mode, which switches to the highvisor, and the host OS handles the interrupt. When running in the host and the highvisor, interrupts go directly to kernel mode, avoiding the overhead of going through Hyp mode. In both cases, all hardware interrupt processing is done in the host by reusing Linux's existing interrupt handling functionality.

However, VMs from time to time require notifications in the form of virtual interrupts from emulated devices and multicore guest OSes must be able to send virtual IPIs from one virtual core to another. KVM/ARM uses the virtual GIC distributor (VGIC) to inject virtual interrupts into VMs to reduce the number of traps to Hyp mode. As described in Section 2.1, virtual interrupts are raised to virtual CPUs by programming the list registers in the GIC hypervisor control interface. KVM/ARM configures the stage 2 page tables to prevent VMs from accessing any other part of the GIC than the virtual CPU interface, including the hypervisor control interface, the physical GIC distributor, and the physical CPU interface. The GIC virtual CPU interface is guaranteed to be contained within a single page and can therefore be mapped to the VM without giving the VM access to any other resources. GICv3 uses system register accesses to the virtual CPU interface and in this case, no part of the GICv3 hardware is mapped for direct MMIO access to the VM. Instead, the CPU interface system register accesses are redirected by the hardware to the virtual CPU interface registers. On GICv3, the hypervisor control interface is controlled through system registers only accessible in Hyp mode. This configuration ensures that for both GICv2 and GICv3, only the hypervisor can program the control interface and that the VM can access the GIC virtual CPU interface directly. However, guest OSes will still attempt to access a GIC distributor to configure the GIC and to send IPIs from one virtual core to another. Such accesses will trap to the hypervisor and the hypervisor must emulate a virtual distributor as part of the VGIC.

KVM/ARM introduces the virtual distributor, a software model of the GIC distributor as part of the highvisor. It is attractive to implement the emulated distributor in the lowvisor because it would potentially allow for faster handling of traps from the VM, but this doesn't work, because

the distributor emulation must be able to use kernel spinlocks, send physical IPIs, and handle events from kernel threads when processing interrupts, and these features are part of the host OS kernel and not available in the lowvisor. The virtual distributor exposes an interface to user space and the rest of the kernel, so that emulated devices in user space can raise virtual interrupts to the virtual distributor. Other parts of the kernel, including VHOST used for the virtual Virtio device backends and VFIO used for direct device assignment can raise virtual interrupts by signaling file descriptors directly between other kernel subsystems and the virtual distributor. KVM/ARM further exposes an MMIO interface to the VM identical to that of the physical GIC distributor. The virtual distributor keeps internal software state about the state of each interrupt and uses this state whenever a VM is scheduled to program the list registers to inject virtual interrupts. For example, if virtual CPU0 sends an IPI to virtual CPU1, the distributor will program the list registers for virtual CPU1 to raise a virtual IPI interrupt the next time virtual CPU1 runs.

Note that the list registers are per physical CPU. This means that a physical CPU wishing to inject a virtual interrupt to a VCPU, which is running on a different physical CPU, must first register the pending virtual interrupt in software and then send a physical IPI from the sending physical CPU to the receiving physical CPU, which is running the VCPU. The physical IPI causes the CPU to exit from the VM and return to the highvisor, where it will check for pending virtual interrupts for the VCPU, and only then will it program its own list registers. Therefore, most virtual interrupt injection scenarios to a running VM involves underlying cross-physical CPU communications including physical IPIs.

Ideally, the virtual distributor only accesses the hardware list registers when necessary, since device MMIO operations are typically significantly slower than cached memory accesses. A complete context switch of the list registers is required when scheduling a different VM to run on a physical core, but not necessarily required when simply switching between a VM and the hypervisor. For example, if there are no pending virtual interrupts, it is not necessary to access any of the list registers. Note that once the hypervisor writes a virtual interrupt to a list register when switching to a VM, it must also read the list register back when switching back to the hypervisor, because the list register describes the state of the virtual interrupt and indicates, for example, if the VM has acknowledged the virtual interrupt. The initial unoptimized version of KVM/ARM used a simplified approach which completely context switched all VGIC state including the list registers on each

transition between the VM and the hypervisor and vice versa. See Chapter 4 for more information on optimizing VGIC performance.

2.2.6 Timer Virtualization

Reading counters and programming timers are frequent operations in many OSes used for process scheduling and to regularly poll device state. For example, Linux reads a counter to determine if a process has expired its time slice and programs timers to ensure that processes don't exceed their allowed time slices. Application workloads also often leverage timers for various reasons. Trapping to the hypervisor for each such operation is likely to incur noticeable performance overheads, and allowing a VM direct access to the time-keeping hardware typically implies giving up timing control of the hardware resources as VMs can disable timers and control the CPU for extended periods of time.

KVM/ARM leverages ARM's hardware virtualization features of the generic timers to allow VMs direct access to reading counters and programming timers without trapping to Hyp mode while at the same time ensuring the hypervisor remains in control of the hardware. While the architecture provides a dedicated Hyp timer in addition to the physical and virtual timers, as described in Section 2.1.4, the Hyp timer is only available in Hyp mode, and therefore not suitable for the host Linux OS's timekeeping needs. Instead, the host Linux operating system uses the physical timer. Since Linux only uses the virtual timer and not the physical timer when running in a VM (it detects that it boots in EL1 instead of EL2), KVM/ARM currently dedicates the virtual timer for the VM and uses the physical timer for the host. Furthermore, KVM/ARM currently disables access to the physical timer when running a VM so that it is reserved for the host OS, and VM accesses to the physical timer are trapped and emulated in software. It is potentially possible to move the physical timer state to the hypervisor timer in the lowvisor when switching to the VM and thereby making both the virtual and physical timer hardware available to VMs when using split-mode virtualization. The Linux kernel running as a guest OS only accessing the virtual timer benefits in performance from being able to directly access timer hardware without trapping to the hypervisor.

Unfortunately, the ARM architecture does not provide any mechanism for directly injecting virtual interrupts from timers to the VM. Timers always generate physical interrupts which trap to the hypervisor. When the VM is running, and the virtual timer expires, the physical interrupt traps

to the lowvisor, which disables the virtual timer as part of switching to the highvisor, and therefore the highvisor never actually handles the interrupt from the virtual timer. When returning from the VM to the highvisor, the highvisor always checks if the virtual timer state for the VCPU has expired and injects a virtual interrupt if it has.

The hardware only provides a single virtual timer per physical CPU, and multiple virtual CPUs may be multiplexed across this single hardware instance. To support virtual timers in this scenario, KVM/ARM uses an approach similar to x86 and detects unexpired timers when a VM traps to the hypervisor and leverages existing OS functionality to program a software timer at the time when the virtual timer would have otherwise fired, had the VM been left running. When such a software timer fires, a callback function is executed, which raises a virtual timer interrupt to the VM using the virtual distributor described above. A common scenario is an idle VCPU scheduling a future timer and then halting the VCPU using the ARM wait-for-interrupt (`WFI`) instruction. Executing WFI in a VM traps to the hypervisor and once again the abstraction of Linux processes and threads to VMs and VCPUs is useful. A WFI instruction executed in a VM is emulated simply by telling the Linux scheduler that the corresponding VCPU thread does not have any work to do. When the software timer expires and the callback function is called, the VCPU thread is woken up and Linux schedules the thread. Upon entering the VM, the highvisor will detect the pending virtual timer for the VCPU, and inject the virtual interrupt.

2.3 Reflections on Implementation and Adoption

The KVM/ARM architecture presented in Section 2.2 was merged into mainline Linux version 3.9 and is now the standard ARM hypervisor on Linux platforms. I share some lessons learned from my experiences in hope that they may be helpful to others in getting research ideas widely adopted by the open source community.

Code Maintainability is Key. An important point that may not be considered when making code available to an open source community is that any implementation must be maintained. If an implementation requires many people and much effort to be maintained, it is much less likely to be integrated into existing open source code bases. Because maintainability is so crucial, reusing code and interfaces is important. For example, KVM/ARM builds on existing infrastructure such as

KVM and QEMU, and from the very start I prioritized addressing code review comments to make my code suitable for integration into existing systems. An unexpected but important benefit of this decision was that I could leverage the Linux community for help to solve hard bugs or understand intricate parts of the ARM architecture.

Be a Known Contributor. Convincing maintainers to integrate code is not just about the code itself, but also about who submits it. It is not unusual for researchers to complain about kernel maintainers not accepting their code into Linux only to have some known kernel developer submit the same idea and have it accepted. The reason is an issue of trust. Establishing trust is a catch-22: one must be well-known to submit code, yet one cannot become known without submitting code. My simple recommendation is to start small. As part of my work, I also made various small changes to KVM to prepare support for ARM, which included cleaning up existing code to be more generic and improving cross platform support. While these patches were small and did not require much work, submitting them showed that I was able to format patches, write a proper commit message in understandable English, understand the project work flow, and cared about the overall quality of the software project. The KVM maintainers at the time were glad to accept these small improvements, which generated goodwill and helped me become known to the KVM community.

Make Friends and Involve the Community. Open source development turns out to be quite a social enterprise. Networking with the community helps tremendously, not just online, but in person at conferences and other venues. For example, at an early stage in the development of KVM/ARM, I traveled to ARM headquarters in Cambridge, UK to establish contact with both ARM management and the ARM kernel engineering team, who both contributed to my efforts.

As another example, an important issue in integrating KVM/ARM into the kernel was agreeing on various interfaces for ARM virtualization, such as reading and writing control registers. Since it is an established policy to never break released interfaces and compatibility with user space applications, existing interfaces cannot be changed, and the Linux community puts great effort into designing extensible and reusable interfaces. Deciding on the appropriateness of an interface is a judgment call and not an exact science. I was fortunate enough to receive help from well-known kernel developers including Marc Zyngier and Rusty Russell, who helped me drive both the implementation and communication about proposed interfaces, specifically for user space save and restore of registers, a feature useful for both debugging and VM migration. Working with established de-

velopers was a tremendous help because we benefited from both their experience and strong voice in the kernel community.

Involve the Community Early. An important issue in developing KVM/ARM was how to get access to Hyp mode across the plethora of available ARM SoC platforms supported by Linux. One approach would be to initialize and configure Hyp mode when KVM is initialized, which would isolate the code changes to the KVM subsystem. However, because getting into Hyp mode from the kernel involves a trap, early stage bootloaders would have had to already have installed code in Hyp mode to handle the trap and allow KVM to run. If no such trap handler was installed, trapping to Hyp mode could end up crashing the kernel. We worked with the kernel community to define the right binary interface (ABI) between KVM and the bootloader, but soon learned that agreeing on ABIs with SoC vendors had historically been difficult.

In collaboration with ARM and the open source community, we reached the conclusion that if we simply required the kernel to be booted in Hyp mode, we would not have to rely on fragile ABIs. The kernel simply tests during boot whether it is running in Hyp mode, in which case it installs a trap handler to provide a hook to re-enter Hyp mode at a later stage. A small amount of code must be added to the kernel boot procedure, but the result is a much cleaner and robust mechanism. If the bootloader is Hyp mode unaware and the kernel does not boot up in Hyp mode, KVM/ARM will detect this and will simply remain disabled. This solution avoids the need to design a new ABI and it turned out that legacy kernels would still work, because they always make an explicit switch into kernel mode as their first instruction. These changes were merged into the mainline Linux 3.6 kernel, and the ARM kernel boot recommendations were modified to recommend that all bootloaders boot the kernel in Hyp mode to take advantage of the new architecture features.

Know the Upstream Project Process. It is crucial to understand the actual process of upstreaming changes to an open-source software project. In the Linux kernel, and many other projects, the software project works through thoroughly established mechanisms, but these may not be very well documented. For example, the Linux kernel is divided into several subsystems, each maintained by its own maintainer(s). The maintainer will accept well-reviewed, tested, and trusted code, and the maintainer is chosen depending on which subsystem the majority of a set of changes affects. Getting maintainers across subsystems to agree on taking responsibility and coordinating a merge effort can be a demanding task, so splitting changes up into segments that are mostly isolated to

individual subsystems can go a long way in making the upstream process itself simpler.

In the Linux kernel, patches are submitted to one or more mailing lists, for example many patches are always cc'ed to the Linux Kernel Mailing List (LKML). Members of the community read the patches, perform code review and test the patches, and report back concerns or problems with the code. The patch submitter then resubmits his/her patches and awaits further comments. When community members are sufficiently happy with the state of the code changes, they will signify this by giving their *reviewed-by* or *acked-by* tags. If maintainers see a patch series which is reviewed or acked by trusted contributors, they can then choose to merge the code with some confidence of the correctness, maintainability, and overall quality of the code. Gathering reviews and acks should therefore be a key goal for any software developer attempting to merge code in the Linux kernel. Many open source projects such as QEMU and GCC have similar processes.

There were multiple possible upstream paths for KVM/ARM. Historically, other architectures supported by KVM such as x86 and PowerPC were merged through the KVM tree directly into Linus Torvalds' tree with the appropriate approval of the respective architecture maintainers. However, KVM/ARM required a few minor changes to ARM-specific header files and the idmap subsystem, and it was therefore not clear whether the code would be integrated via the KVM tree with approval from the ARM kernel maintainer or via the ARM kernel tree. At the time of writing, Russell King is the 32-bit ARM kernel maintainer, and Linus Torvalds pulls directly from his kernel tree for 32-bit ARM related changes. The situation was particularly interesting, because Russell King did not want to merge virtualization support in the mainline kernel [58] and he did not review our code. At the same time, the KVM community was quite interested in integrating our code, but could not do so without approval from the ARM maintainers, and Russell King refused to engage in a discussion about this procedure.

Be Persistent. While we were trying to merge our code into Linux, a lot of changes were happening around Linux ARM support in general. The amount of churn [29] in SoC support code was becoming an increasingly big problem for maintainers, and much work was underway to reduce board specific code and support a single ARM kernel binary bootable across multiple SoCs [60]. In light of these ongoing changes, getting enough time from ARM kernel maintainers to review the code was challenging, and there was extra pressure on the maintainers to be highly critical of any new code merged into the ARM tree. We had no choice but to keep maintaining and improving the

code, regularly sending out updated patch series that followed upstream kernel changes. Eventually Will Deacon, one of the 64-bit ARM kernel maintainers, made time for several comprehensive and helpful reviews, and after addressing his concerns, he gave us his approval of the code by adding an acked-by message to all the individual patches. After all this, when we thought we were done, we received some feedback from the 32-bit ARM maintainer, asking us to rewrite another major part of the ARM kernel to get our patches merged.

When MMIO operations trap to the hypervisor, the virtualization extensions populate a register which contains information useful to emulate the instruction (whether it was a load or a store, source/target registers, and the length of MMIO accesses). A certain class of instructions, those which write back an adjusted memory address to the base register, used by older Linux kernels, do not populate such a register. KVM/ARM therefore loaded the instruction from memory and decoded it in software. Even though the decoding implementation was well tested and reviewed by a large group of people, Russell King objected to including this feature. He had already implemented multiple forms of instruction decoding in other subsystems and demanded that we either rewrite significant parts of the ARM kernel to unify all instruction decoding to improve code reuse, or drop the MMIO instruction decoding support from our implementation. Rather than pursue a rewriting effort that could drag on for months, we abandoned the otherwise well-liked code base. We can only speculate that the true motives behind the decision was to horse-trade the laborious effort of rewriting parts of the ARM kernel in exchange for getting KVM support merged. Unfortunately, the ARM maintainer would not engage in a discussion about the subject.

As a curious epilogue, it turned out that using register-writeback instructions to perform MMIO operations, is not architecturally guaranteed to work as expected, even on physical hardware. Linux was therefore modified to avoid these load/store instruction when accessing device memory, making the missing MMIO instruction decoding a problem only for legacy Linux kernels. Still, proprietary guest operating systems may still emit these instructions and will not work with the 32-bit KVM/ARM implementation. This problem does not exist for 64-bit ARM systems, because the AArch64 execution mode supports decoding all load/store instructions in hardware.

After 15 main patch revisions and more than 18 months, the KVM/ARM code was successfully merged into Linus's tree via Russell King's ARM tree in February 2013. The key in getting all these things to come together in the end before the 3.9 merge window was having a good working

relationship with many of the kernel developers to get their help, and being persistent in continuing to make updated patches available, and continuing to address review comments and improve the code.

Understand the maintainer's point of view. When submitting code to an upstream open-source project it is reviewed by the community as a whole, but most projects have a structure of a maintainer and sub-maintainers. These are the people responsible for certain parts of the code and they will typically have to read all code changes to the files they are responsible for. For active open-source projects, maintainers are busy, often having to review thousands of lines of code per software release, and more and more often the bandwidth of the maintainer becomes a bottleneck for getting code accepted into the open source project. Maintainers in Linux would often tell people requesting certain features that *patches are welcome*, but in reality, maintainers may be too busy to patiently review patches. Maintainers are often overloaded and accepting code form *drive-by* developers who may not participate in working on the software in the long run, means that maintainers take on an even larger burden.

Given the pressure on maintainers, submitting code which is hard to read and understand, where the purpose of the code is not clear, or where the contributor didn't take the time to think carefully about the stability of the code, decreases the likelihood of the code being accepted. It is therefore important that the contributor takes the time to write an explanation of the changes he or she is submitting, for example by writing good documentation and cover letters, and takes responsibility for their work by committing to help fixing problems with the code and explaining difficult parts of the code, even after the code has been merged into the mainline project.

2.4 Experimental Results

This section presents experimental results that quantify the performance of KVM/ARM on multi-core ARMv7 development hardware. These results were used when initially developing and evaluating KVM/ARM and are the first results using ARMv7 hardware with the virtualization extensions. Our results show that split-mode virtualization incurs a higher base cost for transitioning between the VM and the hypervisor, but shows comparable overhead to x86 for real application workloads, generally with less than 10% overhead compared to native execution.

We evaluate the virtualization overhead of KVM/ARM compared to native execution by running both microbenchmarks and real application workloads within VMs and directly on the hardware. We measure and compare the performance, energy, and implementation costs of KVM/ARM versus KVM x86 to demonstrate the effectiveness of KVM/ARM against a more mature hardware virtualization platform.

2.4.1 Methodology

ARMv7 measurements were obtained using an Insignal Arndale board [49] with a dual core 1.7GHz Cortex A-15 CPU on a Samsung Exynos 5250 SoC. This is the first and most widely used commercially available development board based on the Cortex A-15, the first ARM CPU with hardware virtualization support. Onboard 100Mb Ethernet is provided via the USB bus and an external 120GB Samsung 840 series SSD drive was connected to the Arndale board via eSATA. x86 measurements were obtained using both a low-power mobile laptop platform and an industry standard server platform. The laptop platform was a 2011 MacBook Air with a dual core 1.8GHz Core i7-2677M CPU, an internal Samsung SM256C 256GB SSD drive, and an Apple 100Mb USB Ethernet adapter. The server platform was a dedicated OVH SP 3 server with a dual core 3.4GHz Intel Xeon E3 1245v2 CPU, two physical SSD drives of which only one was used, and 1GB Ethernet connected to a 100Mb network infrastructure. x86 hardware with virtual APIC support was not yet available at the time of these experiments.

Given the differences in hardware platforms, our focus was not on measuring absolute performance, but rather the relative performance differences between virtualized and native execution on each platform. Since our goal is to evaluate KVM/ARM, not raw hardware performance, this relative measure provides a useful cross-platform basis for comparing the virtualization performance and power costs of KVM/ARM versus KVM x86.

To provide comparable measurements, we kept the software environments across all hardware platforms as similar as possible. Both the host and guest VMs on all platforms were Ubuntu version 12.10. We used the mainline Linux 3.10 kernel for our experiments, with patches for huge page support applied on top of the source tree. Since the experiments were performed on a number of different platforms, the kernel configurations had to be slightly different, but all common features were configured similarly across all platforms. In particular, Virtio drivers were used in the guest

VMs on both ARM and x86. We used QEMU version v1.5.0 for our measurements. All systems were configured with a maximum of 1.5GB of RAM available to the respective guest VM or host being tested. Furthermore, all multicore measurements were done using two physical cores and guest VMs with two virtual CPUs, and single-core measurements were configured with SMP disabled in the kernel configuration of both the guest and host system; hyperthreading was disabled on the x86 platforms. CPU frequency scaling was disabled to ensure that native and virtualized performance was measured at the same clock rate on each platform.

For measurements involving the network and another server, 100Mb Ethernet was used on all systems. The ARM and x86 laptop platforms were connected using a Netgear GS608v3 switch, and a 2010 iMac with a 3.2GHz Core i3 CPU with 12GB of RAM running Mac OS X Mountain Lion was used as a server. The x86 server platform was connected to a 100Mb port in the OVH network infrastructure, and another identical server in the same data center was used as the server. While there are some differences in the network infrastructure used for the x86 server platform because it is controlled by a hosting provider, we do not expect these differences to have any significant impact on the relative performance between virtualized and native execution.

We present results for four sets of experiments. First, we measured the cost of various micro architectural characteristics of the hypervisors on multicore hardware using custom small guest OSes [53, 31]. We further instrumented the code on both KVM/ARM and KVM x86 to read the cycle counter at specific points along critical paths to more accurately determine where overhead time was spent.

Second, we measured the cost of a number of common low-level OS operations using lmbench [61] v3.0 on both single-core and multicore hardware. When running lmbench on multicore configurations, we pinned each benchmark process to a separate CPU to measure the true overhead of interprocessor communication in VMs on multicore systems.

Third, we measured real application performance using a variety of workloads on both single-core and multicore hardware. Table 2.5 describes the eight application workloads we used.

Fourth, we measured energy efficiency using the same eight application workloads used for measuring application performance. ARM power measurements were performed using an ARM Energy Probe [8] which measures power consumption over a shunt attached to the power supply of the Arndale board. Power to the external SSD was delivered by attaching a USB power cable

apache	Apache v2.2.22 Web server running ApacheBench v2.3 on the local server, which measures number of handled requests per seconds serving the index file of the GCC 4.4 manual using 100 concurrent requests
mysql	MySQL v14.14 (distrib 5.5.27) running the SysBench OLTP benchmark using the default configuration
memcached	memcached v1.4.14 using the `memslap` benchmark with a concurrency parameter of 100
kernel compile	kernel compilation by compiling the Linux 3.6.0 kernel using the vexpress_defconfig for ARM using GCC 4.7.2 on ARM and the GCC 4.7.2 `arm-linux-gnueabi-` cross compilation toolchain on x86
untar	`untar` extracting the 3.6.0 Linux kernel image compressed with bz2 compression using the standard `tar` utility
curl 1K	`curl` v7.27.0 downloading a 1KB randomly generated file 1,000 times from the respective iMac or OVH server and saving the result to `/dev/null` with output disabled, which provides a measure of network latency
curl 1G	`curl` v7.27.0 downloading a 1GB randomly generated file from the respective iMac or OVH server and saving the result to `/dev/null` with output disabled, which provides a measure of network throughput
hackbench	`hackbench` [62] using unix domain sockets and 100 process groups running with 500 loops

Table 2.5: Benchmark Applications

to the USB ports on the Arndale board thereby factoring storage power into the total SoC power measured at the power supply. x86 power measurements were performed using the `powerstat` tool, which reads ACPI information. `powerstat` measures total system power draw from the battery, so power measurements on the x86 system were run from battery power and could only be run on the x86 laptop platform. Although we did not measure the power efficiency of the x86 server platform, it is expected to be much less efficient that the x86 laptop platform, so using the x86 laptop platform provides a conservative comparison of energy efficiency against ARM. The display and wireless features of the x86 laptop platform were turned off to ensure a fair comparison. Both tools reported instantaneous power draw in watts with a 10Hz interval. These measurements were averaged and multiplied by the duration of the test to obtain an energy measure.

Micro Test	ARM	ARM no VGIC/vtimers	x86 laptop	x86 server
Hypercall	5,326	2,270	1,336	1,638
Trap	27	27	632	821
I/O Kernel	5,990	2,850	3,190	3,291
I/O User	10,119	6,704	10,985	12,218
IPI	14,366	32,951	17,138	21,177
EOI+ACK	427	13,726	2,043	2,305

Table 2.6: Micro-Architectural Cycle Counts

2.4.2 Performance and Power Measurements

Table 2.6 presents various micro-architectural costs of virtualization using KVM/ARM and KVM x86. Measurements are shown in cycles instead of time to provide a useful comparison across platforms with different CPU frequencies. We show two numbers for the ARM platform where possible, with and without VGIC and virtual timers support.

Hypercall is the cost of two world switches, going from the VM to the host and immediately back again without doing any work in the host. KVM/ARM takes three to four times as many cycles for this operation versus KVM x86 due to two main factors. First, saving and restoring VGIC state to use virtual interrupts is quite expensive on ARM; the x86 hardware did not yet provide such mechanism. The ARM no VGIC/vtimers measurement does not include the cost of saving and restoring VGIC state, showing that this accounts for over half of the cost of a world switch on ARM. Second, x86 provides hardware support to save and restore state on the world switch, which is much faster. ARM requires software to explicitly save and restore state, which provides greater flexibility, but higher costs. Nevertheless, without the VGIC state, the hypercall costs are only about 600 cycles more than the hardware accelerated hypercall cost on the x86 server platform. The ARM world switch costs have not been optimized and can be reduced further. For example, a small patch eliminating unnecessary atomic operations reduces the hypercall cost by roughly 300 cycles, but did not make it into the mainline kernel until after v3.10 was released. As another example, if parts of the VGIC state were lazily context switched instead of being saved and restored on each world switch, this may also reduce the world switch costs.

Trap is the cost of switching the hardware mode from the VM into the respective CPU mode for running the hypervisor, Hyp mode on ARM and root mode on x86. ARM is much faster than x86 because it only needs to manipulate two registers to perform this trap, whereas the cost of a trap on

x86 is roughly the same as the cost of a world switch because the same amount of state is saved by the hardware in both cases. The trap cost on ARM is a very small part of the world switch costs, indicating that the double trap incurred by split-mode virtualization on ARM does not add much overhead.

I/O Kernel is the cost of an I/O operation from the VM to a device, which is emulated inside the kernel. I/O User shows the cost of issuing an I/O operation to a device emulated in user space, adding to I/O Kernel the cost of transitioning from the kernel to a user space process and doing a small amount of work in user space on the host for I/O. This is representative of the cost of using QEMU for I/O. Since these operations involve world switches, saving and restoring VGIC state is again a significant cost on ARM. KVM x86 is faster than KVM/ARM on I/O Kernel, but slightly slower on I/O User. This is because the hardware optimized world switch on x86 constitutes the majority of the cost of performing I/O in the kernel, but transitioning from kernel to a user space process on the host side is more expensive on x86 because x86 KVM saves and restores additional state lazily when going to user space. Note that the added cost of going to user space includes saving additional state, doing some work in user space, and returning to the kernel and processing the `KVM_RUN` ioctl call for KVM.

IPI is the cost of issuing an IPI from one VCPU to another VCPU when both VCPUs are running on separate physical CPUs and both physical CPUs are running the VM. IPI measures time starting from sending an IPI until the other virtual core responds and completes the IPI. It involves multiple world switches and sending and receiving a hardware IPI. Despite its higher world switch cost, ARM is faster than x86 because the underlying hardware IPI on x86 is expensive, x86 APIC MMIO operations require KVM x86 to perform instruction decoding not needed on ARM, and completing an interrupt on x86 is more expensive. ARM without VGIC/vtimers is significantly slower than with VGIC/vtimers even though it has lower world switch costs because sending, deactivating and acknowledging interrupts trap to the hypervisor and are handled by QEMU in user space.

EOI+ACK is the cost of completing a virtual interrupt on both platforms. It includes both interrupt acknowledgment and deactivation on ARM, but only completion on the x86 platform. ARM requires an additional operation, the acknowledgment, to the interrupt controller to determine the source of the interrupt. x86 does not because the source is directly indicated by the interrupt descriptor table entry at the time when the interrupt is raised. However, the operation is roughly 5

times faster on ARM than x86 because there is no need to trap to the hypervisor on ARM because of VGIC support for both operations. On x86, the EOI operation must be emulated and therefore causes a trap to the hypervisor. This operation is required for every virtual interrupt including both virtual IPIs and interrupts from virtual devices.

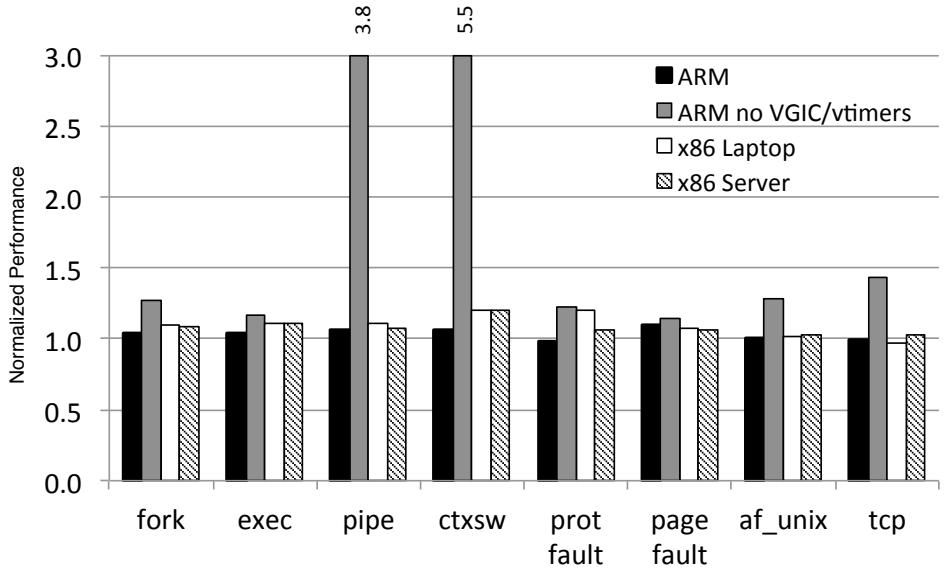

Figure 2.6: UP VM Normalized lmbench Performance

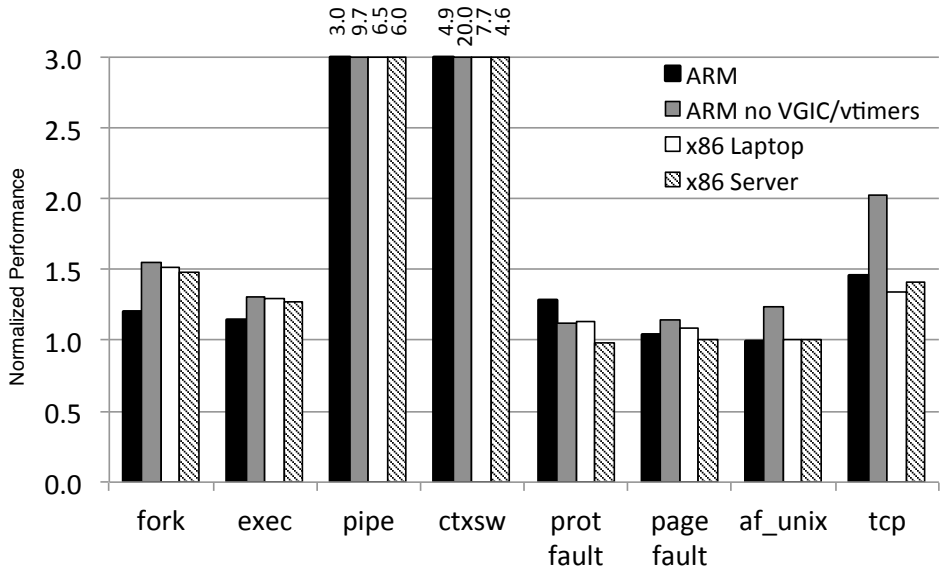

Figure 2.7: SMP VM Normalized lmbench Performance

Figures 2.6 through 2.10 show virtualized execution measurements normalized relative to their

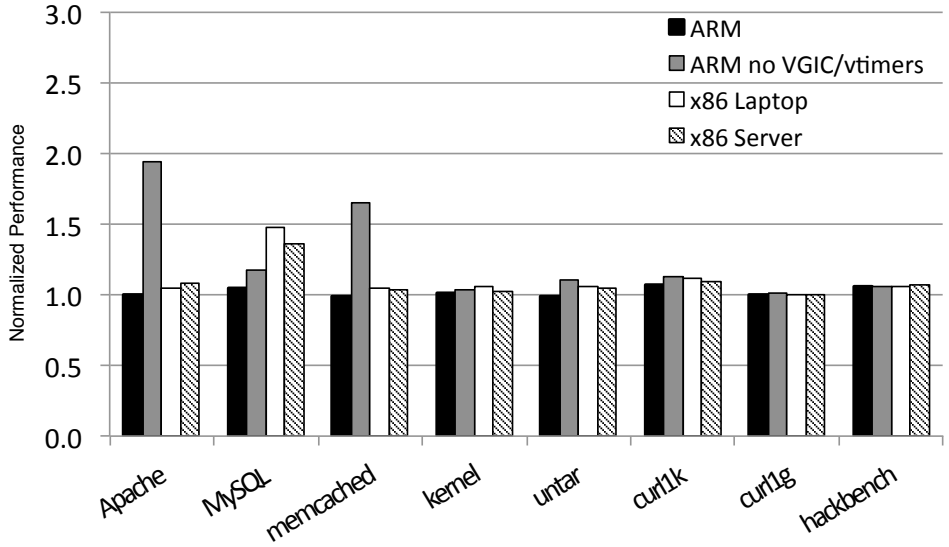

Figure 2.8: UP VM Normalized Application Performance

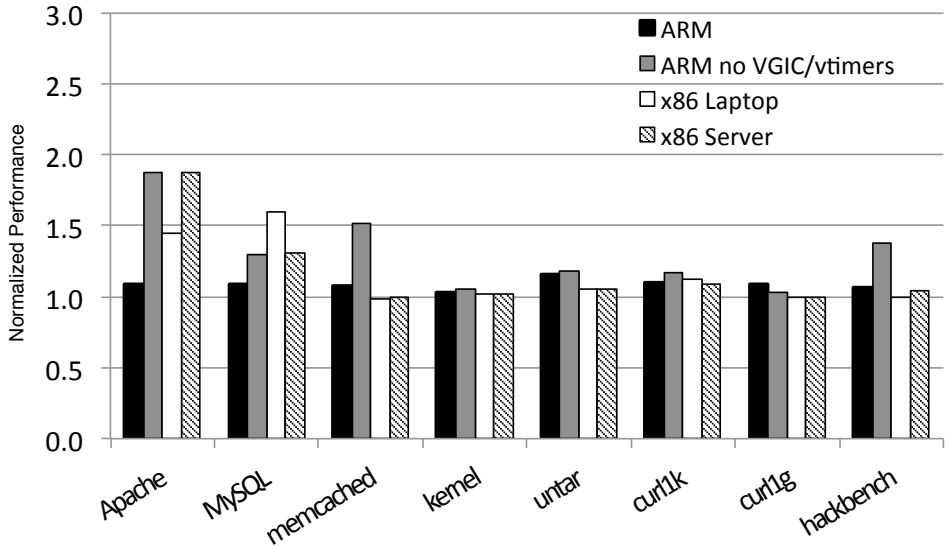

Figure 2.9: SMP VM Normalized Application Performance

respective native execution measurements, with lower being less overhead. Figures 2.6 and 2.7 show normalized performance for running lmbench in a VM versus running directly on the host. Figure 2.6 shows that KVM/ARM and KVM x86 have similar virtualization overhead in a single core configuration. For comparison, we also show KVM/ARM performance without VGIC/vtimers. Overall, using VGIC/vtimers provides slightly better performance except for the pipe and ctxsw

workloads where the difference is substantial. The reason for the high overhead in this case is caused by updating the runqueue clock in the Linux scheduler every time a process blocks, since reading a counter traps to user space without vtimers on the ARM platform. We verified this by running the workload with VGIC support, but without vtimers, and we counted the number of timer read exits when running without vtimers support.

Figure 2.7 shows more substantial differences in virtualization overhead between KVM/ARM and KVM x86 in a multicore configuration. KVM/ARM has less overhead than KVM x86 fork and exec, but more for protection faults. Both systems have the worst overhead for the pipe and ctxsw workloads, though KVM x86 is more than two times worse for pipe. This is due to the cost of repeatedly sending an IPI from the sender of the data in the pipe to the receiver for each message and the cost of sending an IPI when scheduling a new process. x86 not only has higher IPI overhead than ARM, but it must also complete each IPI, which is much more expensive on x86 than on ARM because this requires trapping to the hypervisor on x86 but not on ARM. Without using VGIC/vtimers, KVM/ARM also incurs high overhead comparable to KVM x86 because it then also traps to the hypervisor to acknowledge and deactivate the IPIs.

Figures 2.8 and 2.9 show normalized performance for running application workloads in a VM versus running directly on the host. Figure 2.8 shows that KVM/ARM and KVM x86 have similar virtualization overhead across all workloads in a single core configuration except for the MySQL workloads, but Figure 2.9 shows that there are more substantial differences in performance on multicore. On multicore, KVM/ARM has significantly less virtualization overhead than KVM x86 on Apache and MySQL. Overall on multicore, KVM/ARM performs within 10% of running directly on the hardware for all application workloads, while the more mature KVM x86 system has significantly higher virtualization overheads for Apache and MySQL. KVM/ARM's split-mode virtualization design allows it to leverage ARM hardware support with comparable performance to a traditional hypervisor using x86 hardware support. The measurements also show that KVM/ARM performs better overall with ARM VGIC/vtimers support than without.

Figure 2.10 shows normalized power consumption of using virtualization versus direct execution for various application workloads on multicore. We only compared KVM/ARM on ARM against KVM x86 on x86 laptop. The Intel Core i7 CPU used in these experiments is one of Intel's more power optimized processors. The measurements show that KVM/ARM using VGIC/vtimers

is more power efficient than KVM x86 virtualization in all cases except memcached and untar. Both workloads are not CPU bound on both platforms and the power consumption is not significantly affected by the virtualization layer. However, due to ARM's slightly higher virtualization overhead for these workloads, the energy virtualization overhead is slightly higher on ARM for the two workloads. While a more detailed study of energy aspects of virtualization is beyond the scope of this dissertation, these measurements nevertheless provide useful data comparing ARM and x86 virtualization energy costs.

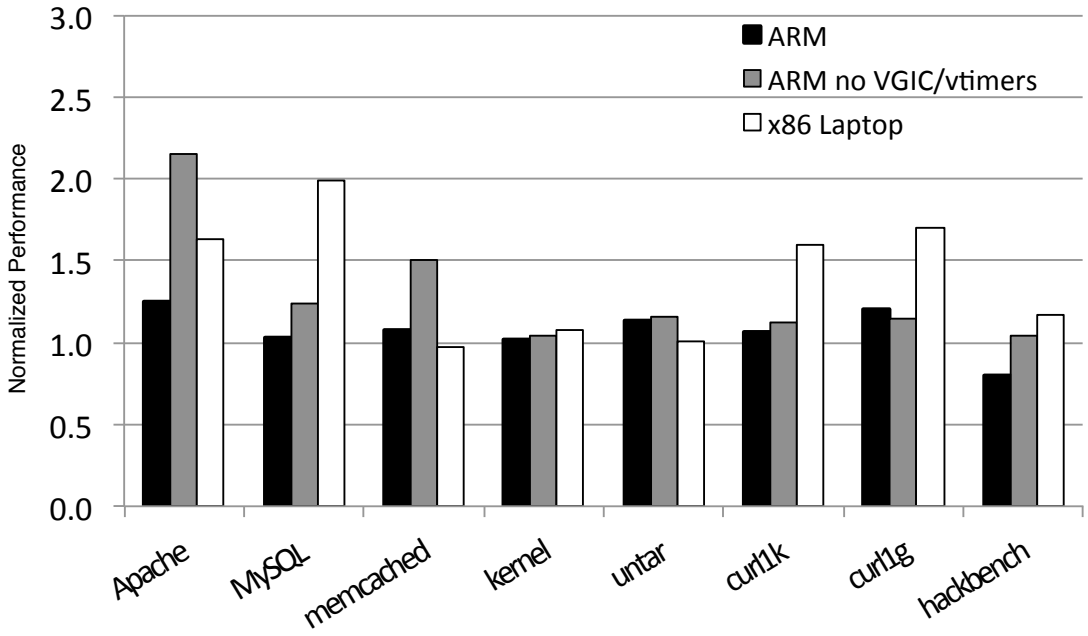

Figure 2.10: SMP VM Normalized Energy Consumption

2.4.3 Implementation Complexity

We compare the code complexity of KVM/ARM to its KVM x86 counterpart in Linux v3.10. KVM/ARM is 5,812 lines of code (LOC), counting just the architecture-specific code added to Linux to implement it, of which the lowvisor is a mere 718 LOC. As a conservative comparison, KVM x86 is 25,367 LOC, excluding 3,311 LOC required for AMD support, and excluding guest performance monitoring and debugging support, not yet supported by KVM/ARM in Linux 3.10. These numbers do not include KVM's architecture-generic code, 7,071 LOC, which is shared by all systems. Table 2.7 shows a breakdown of the total hypervisor architecture-specific code into its major components.

Component	KVM/ARM	KVM x86 (Intel)
Core CPU	2,493	16,177
Page Fault Handling	738	3,410
Interrupts	1,057	1,978
Timers	180	573
Other	1,344	1,288
Architecture-specific	5,812	25,367

Table 2.7: Code Complexity in Lines of Code (LOC)

By inspecting the code we notice that the striking additional complexity in the x86 implementation is mainly due to the five following reasons:

1. Since EPT was not supported in earlier hardware versions, KVM x86 must support shadow page tables.

2. The hardware virtualization support has evolved over time, requiring software to conditionally check for support for a large number of features such as EPT.

3. A number of operations require software decoding of instructions on the x86 platform. Even if we count KVM/ARM's out-of-tree MMIO instruction decode, its implementation was much simpler, only 462 LOC.

4. The various paging modes on x86 requires more software logic to handle page faults.

5. x86 requires more software logic to support timers than ARM, which provides timers hardware support that reduces software complexity.

KVM/ARM's LOC is less than partially complete bare-metal microvisors written for ARM's Hyp mode [81], with the lowvisor LOC almost an order of magnitude smaller. Unlike standalone hypervisors, KVM/ARM's code complexity is so small because lots of functionality simply does not have to be implemented because it is already provided by Linux. Table 2.7 does not include other non-hypervisor architecture-specific Linux code, such as basic bootstrapping, which is significantly more code. Porting a standalone hypervisor such as Xen from x86 to ARM is much more complicated because all of that ARM code for basic system functionality needs to be written from scratch. In contrast, since Linux is dominant on ARM, KVM/ARM just leverages existing Linux ARM support to run on every platform supported by Linux.

2.5 Related Work

The most popular virtualization systems including VMware [24] and Xen [16] were originally based on software-only approaches using either binary translation or paravirtualization [83, 82]. However, today all x86 virtualization systems including VMware [1], Xen, and KVM [54] leverage x86 hardware virtualization support.

Adams and Agesen [1] compared the VMware hypervisor using their software based binary translation with x86 hardware support for virtualization, and found that in the initial version of the x86 hardware support, the software based approach actually outperformed the hardware approach, but they also found that the implementation complexity of the hypervisor is greatly reduced with hardware support for virtualization compared to binary translation techniques.

Some x86 approaches also leverage the host kernel to provide functionality for the hypervisor. VMware Workstation's [77] hypervisor creates a VMM separate from the host kernel, but this approach is different from KVM/ARM in a number of important ways. First, the VMware VMM is not integrated into the host kernel source code and therefore cannot reuse existing host kernel code, for example, for populating page tables relating to the VM. Second, since the VMware VMM is specific to x86 it does not run across different privileged CPU modes, and therefore does not use a design similar to KVM/ARM. Third, most of the emulation and fault-handling code required to run a VM executes at the most privileged level inside the VMM. KVM/ARM executes this code in the less privileged kernel mode, and only executes a minimal amount of code in the most privileged mode. In contrast, KVM benefits from being integrated with the Linux kernel like KVM/ARM, but the x86 design relies on being able to run the kernel and the hypervisor together in the same hardware hypervisor mode, which is problematic on ARM.

Besides being able to support hosted hypervisors using ARM's virtualization extensions, split-mode virtualization also has the potential benefit of running a minimal amount of code in the most privileged CPU mode. This is somewhat reminiscent of microkernels as known from generic operating system design discussions. Microkernel approaches for hypervisors [76, 44] have been used to reduce the hypervisor TCB and run other hypervisor services in user mode. These approaches differ both in design and rationale from split-mode virtualization, which splits hypervisor functionality across privileged modes to leverage virtualization hardware support. Where microkernel

approaches provide abstracted services designed to be used by a mixture of VMs and user space services, split-mode virtualization only provides two runtime functions in its ABI: One to run the VM, and one that returns from the VM. Where microkernel approaches typically run a plethora of user space services that communicate via Inter-Process Communication (IPC) to support the hardware and overall system, split-mode virtualization benefits from the performance, simplicity, and existing support of a widely used monolithic kernel and only needs a memory-mapped interface to support its two interface function between the lowvisor and highvisor. Split-mode virtualization also provides a different split of functionality compared to microkernels, in that it splits the execution of the hypervisor functionality. KVM/ARM's lowvisor only implements the lowest level hypervisor mechanisms and has a much smaller code base than typical microkernels.

After the early effort to support Xen on ARM was abandoned, a new clean slate implementation targeting servers [26] was developed and merged with the x86 Xen code. This new effort is based exclusively on ARM hardware virtualization support, and does not require paravirtualization to support CPU or memory virtualization. Since Xen is a standalone hypervisor, porting Xen from x86 to ARM is difficult in part because all ARM-related code must be written from scratch. Even after getting Xen to work on one ARM platform, it must also be manually ported to each different ARM device that Xen wants to support. Because of Xen's custom I/O model using hypercalls from VMs for device emulation on ARM, Xen unfortunately cannot run guest OSes unless they have been configured to include Xen's hypercall layer and include support for Xen's paravirtualized drivers. Xen can, however, run entirely in Hyp mode rather than using split-mode virtualization, potentially allowing reduced world switch times on ARM systems using the virtualization extensions compared to hosted hypervisors on ARM prior to the work described in Chapter 4. In contrast, our work on KVM/ARM reuses existing Linux support for ARM and standard Linux components to enable faster development. KVM/ARM also supports device emulation, and for example fully emulates the Versatile Express development board and the Calxeda Midway in upstream QEMU. As a result, KVM/ARM can run fully unmodified guest OSes. KVM/ARM is easily supported on new devices with Linux support, and we spent almost no effort beyond the initial implementation to support KVM/ARM on a wide range of ARM development hardware and servers.

2.6 Summary

In this chapter I introduced split-mode virtualization, which makes it possible to use ARM hardware virtualization extensions with a hosted hypervisor design. KVM/ARM uses split-mode virtualization and leverage the Linux kernel's hardware support and functionality to support KVM/ARM's hypervisor functionality. KVM/ARM is the mainline Linux ARM hypervisor and the first system that ran unmodified guest operating systems on ARM multicore hardware.

Our experimental results show that KVM/ARM incurs minimal performance impact from the extra traps incurred by split-mode virtualization, has modest virtualization overhead and power costs, within 10% of direct native execution on multicore hardware for real application workloads, and achieves comparable or lower virtualization overhead and power costs on multicore hardware compared to widely-used KVM x86 virtualization.

Based on our experiences integrating KVM/ARM into the mainline Linux kernel, I provided hints on getting research ideas and code adopted by open source communities.

Chapter 3

Performance of ARM Virtualization

A growing number of companies are now deploying commercially available 64-bit ARMv8 servers to meet their computing infrastructure needs. Major virtualization players, including KVM [37] and Xen [26], leverage the ARM hardware virtualization extensions to support their hypervisor functionality, and ARM virtualization is deployed across a wide range of computing scenarios, from cloud server offerings [64, 73] to locomotive computer systems [20]. Despite these trends and the important of ARM virtualization, little was known in practice regarding how well virtualized systems perform using ARM. 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. Without clear performance data on production server hardware, the ability of hardware and software architects to build efficient ARM virtualization solutions would be severely limited, and companies would not be able to evaluate how best to deploy ARM virtualization solutions to meet their infrastructure needs.

This chapter presents an in-depth study of 64-bit ARMv8 virtualization performance on multi-core server hardware. We measure the performance of the two most popular ARM hypervisors, KVM/ARM 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. KVM/ARM is a hosted hypervisor using split-mode virtualization as presented in Chapter 2, and Xen is a standalone hypervisor.

In contrast to the measurements presented in Chapter 2, which were focused on evaluating the cost of split-mode virtualization on 32-bit ARMv7 development hardware used to initially develop KVM/ARM, the measurement study presented in this chapter evaluates real server application workloads running on 64-bit ARMv8 production server hardware, which was not present when KVM/ARM was originally implemented. This study also considers I/O performance for clients and servers running on ARMv8 server hardware interconnected with fast 10 GbE, and compares and

analyzes the performance of both KVM and Xen on both ARM and x86. The results shown in Chapter 2 show that the cost of performing extra traps in the context of split-mode virtualization when transitioning between the VM and the hypervisor is relatively low, but this chapter expands on this result and shows that the true cost of split-mode virtualization comes from having to save and restore the CPU's register state.

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 standalone and hosted 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 standalone hypervisors, such as Xen, can transition between the VM and the hypervisor 20 times faster than hosted hypervisors, such as KVM, on ARM. We show that ARM can enable significantly faster transitions between the VM and a standalone hypervisor compared to x86. On the other hand, hosted 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, standalone and hosted hypervisors perform equally fast on ARM.

We show that VM application performance is not easily correlated with low-level virtualization microbenchmarks. In fact, for many application workloads, we show that KVM/ARM, a hosted hypervisor, can meet or exceed the performance of Xen ARM, a standalone hypervisor. 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.

The performance characteristics presented in this chapter led to ARM evolving their architecture to provide faster VM-to-hypervisor transitions for hosted hypervisors on ARMv8, and Chapter 4 shows how we leverage these results to improve the performance of KVM/ARM.

3.1 Hypervisor Design

There are two fundamentally different approaches to hypervisor software design. Both hypervisor designs can use trap-and-emulate, paravirtualization, binary translation, or rely on hardware virtual-

ization support. Each hypervisor design, however, can be affected in different ways by the underlying architectural support for virtualization. In the following I briefly explain the design philosophy behind the two designs and how they are affected by ARM's hardware support for virtualization, which are described in more detail in Chapter 2.

3.1.1 Hypervisor Overview

Figure 3.1 depicts the two main hypervisor designs, Type 1 and Type 2, compared to native execution. Type 1 hypervisors, such as 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 1 hypervisors are also known as standalone hypervisors and Type 2 hypervisors are known as hosted hypervisors. Hosted 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 [54] where other solutions such as VMware Workstation [24] use a loadable driver in the existing OS kernel to monitor virtual machines. The OS integrated with a hosted hypervisor is commonly referred to as the host OS, as opposed to the guest OS which runs in a VM.

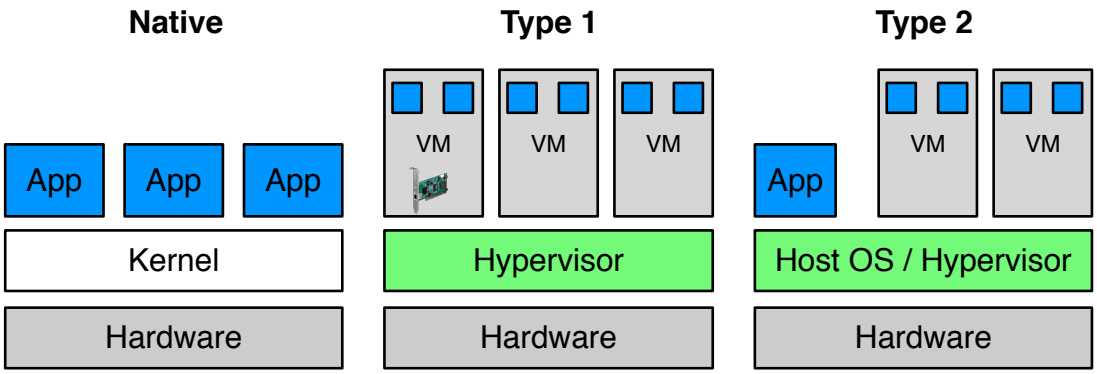

Figure 3.1: Hypervisor Design

One advantage of hosted hypervisors over standalone 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 standalone hypervisor suffers from having to re-implement device drivers for all supported hardware. However, Xen [16], a standalone 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 paravirtualized I/O, as discussed in Section 2.2.4. KVM uses an implementation of the Virtio [70] protocol for disk and networking support, and Xen uses its own implementation referred to simply as *Xen PV*. In KVM, the Virtio-based virtual device backend was originally designed to run in user space (as part of QEMU), but to improve performance the data path backend of Virtio devices were moved into the kernel as part of the Virtio vhost design (see Section 2.2.4 for more info). Therefore, in modern KVM deployments the virtual device backend is implemented in the host OS kernel, and in Xen the virtual device backend is implemented in the Dom0 kernel. A 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.

3.1.2 ARM Hypervisor Implementations

In Section 2.1 we discussed the ARM virtualization extensions and how they provide support for running hypervisors and compared them to x86 hardware virtualization support. This section describes how KVM/ARM and Xen on ARM use the virtualization extensions in each their unique way.

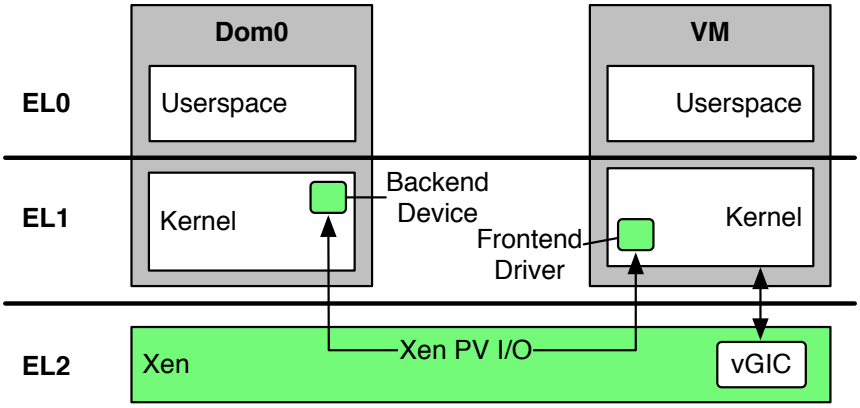

Figure 3.2: Xen ARM Architecture

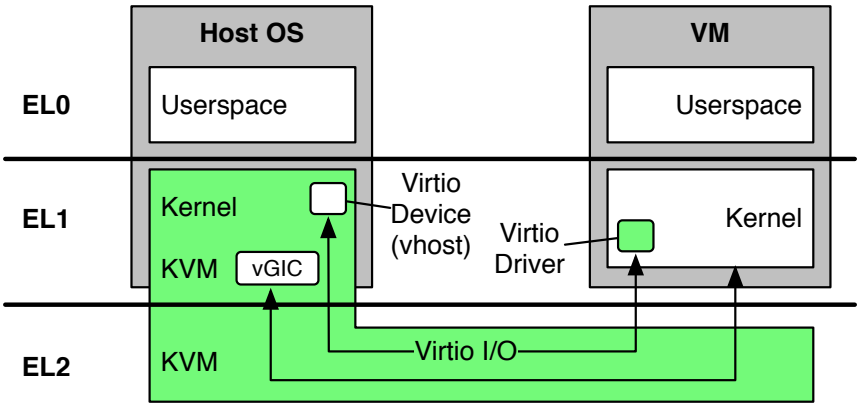

Figure 3.3: KVM/ARM Architecture

As shown in Figures 3.2 and 3.3, Xen and KVM take different approaches to using ARM hardware virtualization support. Xen as a standalone hypervisor design maps easily to the ARM architecture, running the entire hypervisor in EL2 and running VM user space and VM kernel in EL0 and EL1, respectively. However, existing OSes are designed to run in EL1, and a hosted hypervisor that leverages an existing OS such as Linux to interface with hardware does not map as easily to the ARM architecture. EL2 is a strictly more privileged and 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 as described in Section 2.2. Split-mode virtualization shares EL1 between the host OS and VMs and runs a minimal set of hypervisor functionality in EL2 to be able to leverage the ARM virtualization extensions. 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 in concept to a regular process context switch. As we shall see later, this aspect of the architectural support for virtualization has a material impact on the VM-to-hypervisor transition cost.

The difference in how Xen and KVM maps to CPU modes on ARM has no counterpart on x86, because of how the architecture is designed. On x86, the hypervisor runs in root mode, which does not in any way limit or change how CPU privilege levels are used. Running a full featured rich OS like Linux in root mode does not require any changes to the design of the OS, and as a result hosted hypervisors like KVM map just as easily to the x86 architecture as standalone hypervisors, because the entirety of the hypervisor can run in root mode, including the KVM host Linux kernel and the Xen hypervisor.

On ARM, 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 devices 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. As we shall see, the difference between the I/O model of the two hypervisor designs and the cost of the transition sequence described here will have a material impact on the performance of latency sensitive workloads running inside VMs.

3.2 Experimental Design

To evaluate the performance of ARM virtualization, we ran both microbenchmarks and real application workloads on the two 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 [63] 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 has 64 GB of RAM, a 120 GB SATA3 SSD for storage, and a Dual-port Mellanox ConnectX-3 10 GbE NIC. x86 measurements will be 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 has 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 were connected via 10 GbE. and the interconnecting network switch [46] easily handled multiple sets of nodes communicating with full 10 Gb bandwidth such that experiments involving networking between two nodes could 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 user space, 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 use paravirtualized I/O, typical of cloud infrastructure deployments such as Amazon EC2, instead of device passthrough, due to the absence of an IOMMU in available test environments.

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 the precision of our measurements and for our experimental setup to mimic recommended configuration best practices [75], 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.

3.3 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 of 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, but instead becomes overhead compared to native execution. Therefore, the 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 and implemented 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 the 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 3.2, and also ran these measurements pinned to specific VCPUs within VMs with all virtual interrupts assigned to other VCPUs.

Using this framework, we ran seven microbenchmarks that measure various low-level aspects of hypervisor performance. The microbenchmarks are listed and explained in Table 3.1.

Table 3.2 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.

The Hypercall microbenchmark result shows that transitioning from a VM to the hypervisor on ARM can be significantly faster than on 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 3.1, the ARM architecture provides a separate CPU mode with its own register bank to run an isolated standalone hypervisor like Xen. Transitioning from a VM to a standalone 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 result also shows that transitioning from a VM to the hypervisor is more than an order of magnitude more expensive for hosted hypervisors like KVM than for standalone hypervisors like Xen. This is because although all VM traps are handled in

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.
Interrupt Controller Trap	Trap from the VM to the emulated interrupt controller then return to the VM. Measures a frequent operation for many device drivers and baseline for accessing I/O devices emulated in the hypervisor.
Virtual IPI	Issues a virtual IPI from a VCPU to another VCPU running on a different PCPU, both PCPUs executing the VM. Measures time between sending the virtual IPI until the receiving VCPU handles it, a frequent operation in multi-core OSes.
Virtual IRQ Completion	Acknowledges and completes a virtual interrupt inside a VM. On ARM these are two separate operations, ACK and EOI, and on x86 this is a single operation, EOI. Measures a frequent operation in VMs; happens for every injected virtual interrupt.
VM Switch	Switches from one VM to another on the same physical core. Measures a central cost when for example oversubscribing physical CPUs or when switching between an idle domain and a real domain on Xen.
I/O Latency Out	Measures the 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. Measures time passed from the driver in the VM sends the signal until the virtual device backend receives the signal and is able to react to it.
I/O Latency In	Measures the latency between the virtual I/O device in the hypervisor signaling the VM and the VM receiving the corresponding virtual interrupt. The VM is running when the signal is generated. For KVM, a separate thread 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. In both cases, the microbenchmark causes a physical IPI between two PCPUs and the receiving VCPU to exit the VM and inject a virtual interrupt.

Table 3.1: Microbenchmarks

EL2, a hosted hypervisor is integrated with a host kernel and both run in EL1. This results in four additional sources of overhead:

1. 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 3.3, incurring a double trap cost.

	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 3.2: Microbenchmark Measurements (cycle counts)

2. 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 hosted hypervisor host OS, incurring added cost of saving and restoring EL1 register state.

3. 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.

4. Because the hosted 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. The version of KVM/ARM used for these measurements 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 discussed in Section 2.4, our measurements show that there is a substantial cost associated with saving and restoring register

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 3.3: KVM/ARM Hypercall Analysis (cycle counts)

state to switch between EL2 and the host in EL1. Table 3.3 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 additional 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 standalone and hosted 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 3.2. 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 3.2 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 3.2 shows that for the remaining microbenchmarks, 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 [50].

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. VM Switch for KVM x86 is more expensive than the hypercall cost, because KVM x86 performs unloads the active VMCS

from the CPU and loads a new one, which adds overhead, and because it processes deferred work like context switching floating point registers state, which is not necessary on every switch between the VM and the hypervisor. VM Switch for Xen x86 is even more expensive, because Xen spends a large amount of time processing softirqs and other deferred tasks when entering a new VM. We suspect this is an area where the Xen x86 implementation could be improved.

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. Standalone 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 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 3.2, so the qualitative results are not specific to the configuration. We hypothesize that there is no benefit in scheduling Dom0 and the application VM on the same physical core, because

instead of having to perform a full VM switch between Dom0 and the idle domain, Xen now has to switch between the application VM and Dom0 and the workload becomes completely serialized. We further hypothesize that Xen could benefit from optimizing its idle handling to potentially leave the last running VM state on the physical CPU, even when going idle, in the event that the same VCPU will be scheduled again later.

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 3.3.

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.

3.4 Application Benchmark Results

We also 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 3.4 lists the application workloads we use, which include a mix of widely-used CPU and I/O intensive benchmark workloads. For workloads involving a client and a server, we run the client on a dedicated machine and the server on the configuration being measured, ensuring that the client is never saturated during any of our experiments. We run these workloads natively and on both KVM and Xen on both ARM and x86, the latter to provide a baseline comparison.

Kernbench	Compilation of the Linux 3.17.0 kernel using the allnoconfig for ARM using GCC 4.8.2.
Hackbench	`hackbench` [62] using Unix domain sockets and 100 process groups running with 500 loops.
SPECjvm2008	`SPECjvm2008` [74] 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 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 3.4: Application Benchmarks

Given the differences in hardware platforms, our focus is not on measuring absolute performance, but rather the relative performance differences between virtualized and native execution on each platform. Figure 3.4 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. I reported this to the Xen developer com-

munity, and learned that this may be a Mellanox network driver bug exposed by Xen's I/O model. I also reported the issue to the Mellanox driver maintainers, but did not arrive at a solution. Table 3.5 shows the non-normalized results from our measurements.

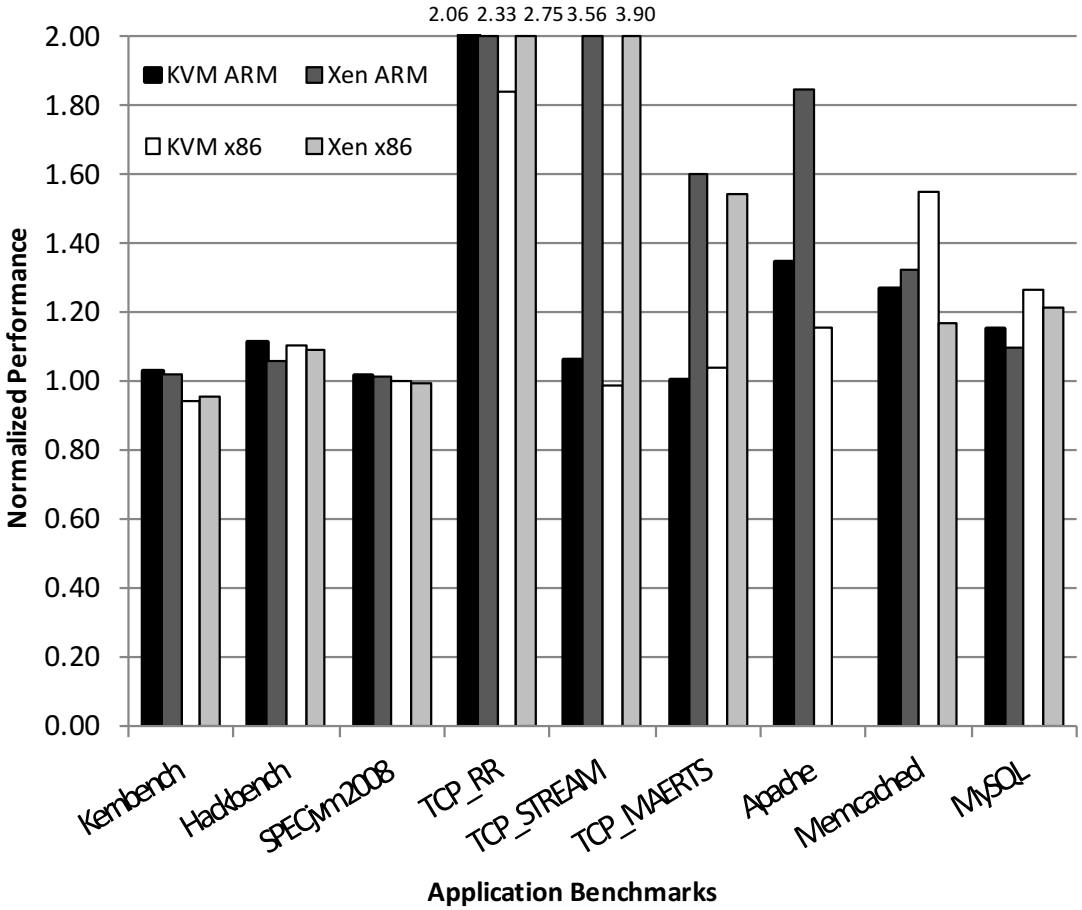

Figure 3.4: Application Benchmark Performance

Figure 3.4 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 3.2. 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.

		Native	KVM	Xen
Kernbench (s)	ARM	49.11	50.49	49.83
	x86	28.91	27.12	27.56
Hackbench (s)	ARM	15.65	17.38	16.55
	x86	6.04	6.66	6.57
SPECjvm (ops/min)	ARM	62.43	61.69	61.91
	x86	140.76	140.64	141.80
TCP_RR (trans/s)	ARM	23,911	11,591	10,253
	x86	21,089	11,490	7,661
TCP_STREAM (Mb/s)	ARM	5,924	5,603	1,662
	x86	9,174	9,287	2,353
TCP_MAERTS (Mb/s)	ARM	6,051	6,059	3,778
	x86	9,148	8,817	5,948
Apache (trans/s)	ARM	6,526	4,846	3,539
	x86	10,585	9,170	N/A
Memcached (ops/s)	ARM	110,865	87,811	84,118
	x86	263,302	170,359	226,403
MySQL (s)	ARM	13.72	15.76	15.02
	x86	7.21	9.08	8.75

Table 3.5: Application Benchmark Raw 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 3.4 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 [43] 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 3.6 shows the detailed measurements.

Table 3.6 shows that the time per transaction increases significantly from 41.8 μs when running natively to 86.3 μs 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 3.3.

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 3.6 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 3.6 also shows that Xen spends noticeably more time than KVM in delivering packets

	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 3.6: Netperf TCP_RR Analysis on ARM

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 3.3. 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 [70], such that the network device can DMA the data directly into a guest-visible buffer, resulting in significantly less overhead.

Furthermore, previous work [72] 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 [48]. 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 [59]. 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 standalone hypervisor like Xen is much lower on ARM than for a hosted hypervisor like KVM, this difference is not easily observed for the application workloads. The reason is that standalone 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 standalone hypervisors on ARM, confirmed by the Interrupt Controller Trap and Virtual IPI microbenchmarks shown in Section 3.3. 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, standalone 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 standalone 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 hosted hypervisor approach of switching to the host on KVM. Additionally, KVM/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/ARM and Xen ARM 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/ARM and Xen ARM can provide virtualization overhead similar to, and in some cases better than, their respective x86 counterparts.

3.5 Related Work

Much work has been done on analyzing and improving the performance of x86 virtualization. While ARM virtualization is a relatively new area it is interesting to note that there are also very little work looking at virtualization performance across different architectures. 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 such as paravirtualization or binary translation. Since both ARM and x86 virtualization today is largely based on hardware assisted virtualization, lessons learned on interactions between the architectural support for virtualization and hypervisor design are of limited value today. Second, some later work based on x86 hardware virtualization support leverages hardware features that are in many cases substantially different from ARM and simply cannot be directly applied on ARM.

One earlier study attempted to estimate the performance of ARM hardware virtualization support using a software simulator [81]. However, the work was based on a simple hypervisor lacking important features like SMP support and use of storage and network devices by multiple VMs. Because of the lack of hardware or a cycle-accurate simulator, no real performance evaluation was possible. In contrast, we have presented an in-depth performance study of ARM virtualization on real 64-bit ARMv8 server hardware using both custom designed micro benchmarks and real-world macro benchmarks measuring multiple production quality hypervisors.

Adams and Agesen [1] compare the performance of a purely software-based hypervisor that uses binary translation with a hypervisor using the first version of x86 hardware virtualization support. They find that the software-based hypervisor outperforms the hardware-based one, but a careful performance analysis shows that this is mostly due to the software hypervisor handling page faults more quickly that the hardware hypervisor. This first iteration of x86 hardware virtualization support did not include virtual MMU support and therefore even the hardware hypervisor still

had to use shadow page table techniques. A later study by Agesen et al. [3] shows that as hardware matured and x86 virtualization support added virtual MMU support (EPT), hardware-based hypervisors outperform software-based hypervisors.

Sugerman et al. [77] optimized I/O performance on an early version of VMware Workstation which was developed before the x86 hardware virtualization support. They apply a number of techniques which are specific to the VMware Workstation hypervisor design. Similar to our studies they find that interrupt processing and running two network stacks are important sources of performance overhead. Their work does not explore design or performance consequences of the x86 architectural support for virtualization or contrast this to any other architectures.

Santos et al. [72] investigate the CPU cost for Netperf's TCP_STREAM benchmark on Xen on x86. Their studies are done on x86 systems without hardware virtualization support using Xen paravirtualization. They find that on such systems, CPU usage per packet increases significantly in Xen compared to bare-metal especially for 10G Ethernet, and suggest changes to the Xen software design to reduce the CPU cost per package. Their experimental setup configures VMs with a single VCPU and does not account for any effects of multicore VMs, commonplace in today's deployments. While Santos et al. focus exclusively on CPU usage for a single streaming benchmark on Xen on x86 using paravirtualization, we have looked at the performance of a wide range of micro benchmarks and application benchmarks, and we contrast different hypervisor designs against each other from a performance point of view in the context of modern hardware support for virtualization.

Heo and Taheri [45] analyze the performance of various latency sensitive workloads on VMware Vsphere on x86. They find that the overhead for latency sensitive workloads is split into a constant part coming from the extra I/O layers and from going through the hypervisor, and a variable part coming from processing the workload in the server. Their main observation is that the variable part is significant and increases proportionally to the size of the server workload. In contrast, we found that on ARM there is very little overhead in the variable workload and that almost the same time is spent executing the workload inside the VM as on bare-metal, and that almost all the overhead comes from the additional I/O layers and from interactions with the hypervisor.

Buell et al. [23] investigate the performance of both complex and latency sensitive workloads on VMware Vsphere on multicore systems using generally available performance tools and using VMware-specific performance tools. Similar to Heo and Taheri they also find that a large part of the

application overhead comes from taking longer to execute the workload in a VM compared to native execution, and not from VM-to-hypervisor transitions. More specifically, they find that the main reason for overhead running complex workloads like OLTP processing comes from the hardware's support for memory virtualization using two-level page tables. In contrast, we found that almost all the overhead from running MySQL on ARM was time spent in the hypervisor.

Common to both of the x86 performance studies referenced above is that the overhead of executing the same code in the VM as on bare-metal comes from memory virtualization. Bhargava et al. [19] also find that two-level TLB misses can be very expensive for any hypervisor using either Intel's or AMD's hardware support for virtualization and recommend that such overhead can be significantly reduced by improving the cache hit rate for page table walks by for example using a larger page granularity for nested pages, or by using a split TLB for first and second stage page table lookups. The ARM architecture allows implementations to use both of these methods and also supports caching intermediate stages of page table walk. The Applied Micro Atlas SoC used in our performance study [33] and presented in Chapter 3 uses a split TLB design and the ARM Cortex-A57 CPU used in the passthrough configuration discussed in Chapter 4 implements caches for intermediate page table walks. In both cases, we did not observe a significant slowdown caused by page table walks as long as we used 2M mappings for all stage 2 mappings on both KVM/ARM and Xen ARM.

3.6 Summary

This chapter presents the first study of ARM virtualization performance on 64-bit ARMv8 server hardware, including multi-core measurements of the two main ARM hypervisors, KVM/ARM and Xen for ARM. To do this, we introduce a suite of microbenchmarks to measure common hypervisor operations on multi-core systems. Using this benchmark suite, we show that ARM enables standalone 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 hosted 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 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 expand on the conclusions and measurements in Chapter 2, which evaluated split-mode virtualization on ARMv7 development hardware. While our previous results show that performing extra traps in the context of split-mode virtualization are relatively inexpensive, this chapter shows that the cost associated with saving and restoring all of the CPU's EL1 kernel register state results in a higher VM-to-hypervisor transition cost for hosted hypervisors using split-mode virtualization compared to standalone hypervisors on ARM.

We further show that both current hypervisor designs cannot leverage ARM's 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 both hypervisor software design and implementation that play a larger role than how quickly the system can switch between the VM and the hypervisor. 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. Finally, we show that ARM hypervisors have similar overhead to their x86 counterparts on real applications.

The results presented in this chapter motivate the introduction of the Virtualization Host Extensions (VHE) for ARMv8, which are designed to improve the performance of hosted hypervisors. In the next chapter I will discuss software changes required to take advantage of these new hardware features which bring faster VM-to-hypervisor transitions to hosted virtualization on ARM and improves real application performance involving I/O.

Chapter 4

Improving ARM Virtualization Performance

In the previous Chapter 3 we have discussed ARM virtualization performance across multiple hypervisors and seen that ARM virtualization has very different performance characteristics compared to x86 virtualization, and that key VM-to-hypervisor interactions perform worse on hosted hypervisors than on standalone hypervisors on ARM. We also saw that hosted hypervisors actually perform better than standalone hypervisors for application workloads involving I/O. This chapter presents a new hypervisor design which brings the benefit of fast VM-to-hypervisor interactions to hosted hypervisors on ARM. We demonstrate the effectiveness of this hypervisor design for KVM/ARM and show that we can significantly improve performance compared to split-mode virtualization.

Hypervisor designs for both ARM and x86 virtualization rely on running a full OS kernel to support the hypervisor functionality. This is true for both standalone and hosted hypervisors. KVM/ARM, a hosted 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 standalone 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 [25]. 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 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), known as VM entries and VM exits for for transitioning to and from the VM, respectively. 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 as shown in Chapter 3.

To address this problem, we present a new hypervisor design and implementation of KVM/ARM that takes advantage of unique features of the ARM architectural support for virtualization in the context of hosted hypervisors. We take a new approach to hypervisor design that runs the hypervisor together with its hypervisor 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.

ARM introduced new architectural features, the Virtualization Host Extensions (VHE), in the ARMv8.1 revision of the architecture, which allows running full unmodified OS kernels in the hypervisor CPU mode, EL2. VHE expands the functionality of EL2 so that it is functionally equivalent to EL1 and provides transparent register redirection from EL1 to EL2 system registers. Our new hypervisor design benefits from the architectural improvements introduced by VHE. With VHE, our design does not require any changes to existing OS outside of the hypervisor functionality. Without VHE, our design requires modifications to the hypervisor OS kernel so it can run in EL2 instead of EL1. Although standalone 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 stan-

dalone hypervisors. We focus on improving the performance of hosted hypervisors on ARM given their widespread popularity, which is at least in part due to their benefits over standalone hypervisors on ARM. ARM hardware does not have the same legacy and standards as x86, so standalone hypervisors have to be manually ported to every hardware platform they support. Hosted 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 design and implementation, resulting in better virtualization performance. First, transitioning from the VM to the hypervisor no longer requires saving and restoring the kernel mode register state. Second, as in the case on ARM, the hypervisor CPU mode can be designed to always have unfettered access to the underlying hardware, and therefore virtualization features such as nested paging do not have to be explicitly enabled and disabled when transitioning between the VM and hypervisor OS kernels. Third, the hypervisor no longer needs to prepare VM state and configuration settings in intermediate data structures when servicing the VM, but can program the hardware state used by the VM directly. For example, x86 hypervisors using VMX must read and write VM state to a special VM Control Structure (VMCS) memory region, and later the VM entry hardware operation copies this state into CPU registers. As another example, KVM/ARM maintains hypervisor-specific data structures in memory while running in EL1 and later copies this data into CPU registers when running in EL2. With our new hypervisor design, KVM/ARM can simply program CPU registers when necessary and otherwise leave them alone, leading to overall improved performance. Also, 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 v4.15, with the remaining changes planned to be applied in Linux v4.16. 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.

4.1 Architectural Support for Hosted Hypervisors

I first provide a brief overview of current state-of-the-art hosted hypervisor designs on both x86 and ARM and discuss how they multiplex kernel mode to run both their VM and hypervisor OS kernels using hardware virtualization support. Chapter 2 describes ARM VE in detail and contrasts them to x86. Here I provide a more thorough explanation of Intel VMX and how it supports multiplexing of kernel mode for x86. I focus on Intel VMX and do not cover AMD-V, though AMD-V is similar for the purposes of this discussion. I also discuss in more detail how ARM VE supports multiplexing of kernel mode between the hypervisor and VM OS kernels.

4.1.1 Intel VMX

The Intel Virtual Machine Extensions (VMX) [50], 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.

4.1.2 ARM VE

As explained in Section 2.1, ARM took a very different approach than x86 in adding hardware virtualization support, and adds a dedicated CPU mode called EL2 to run the hypervisor. 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 user space 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, which 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 user space 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 the x86 VMX transitions, 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 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.

4.1.3 KVM

Figure 4.1 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 4.1(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. In contrast, KVM/ARM uses split-mode virtualization as introduced in Chapter 2, to support both the host OS kernel running in EL1 and at the same time runs software in EL2 to manage the virtualization features and multiplex EL1. Most of the

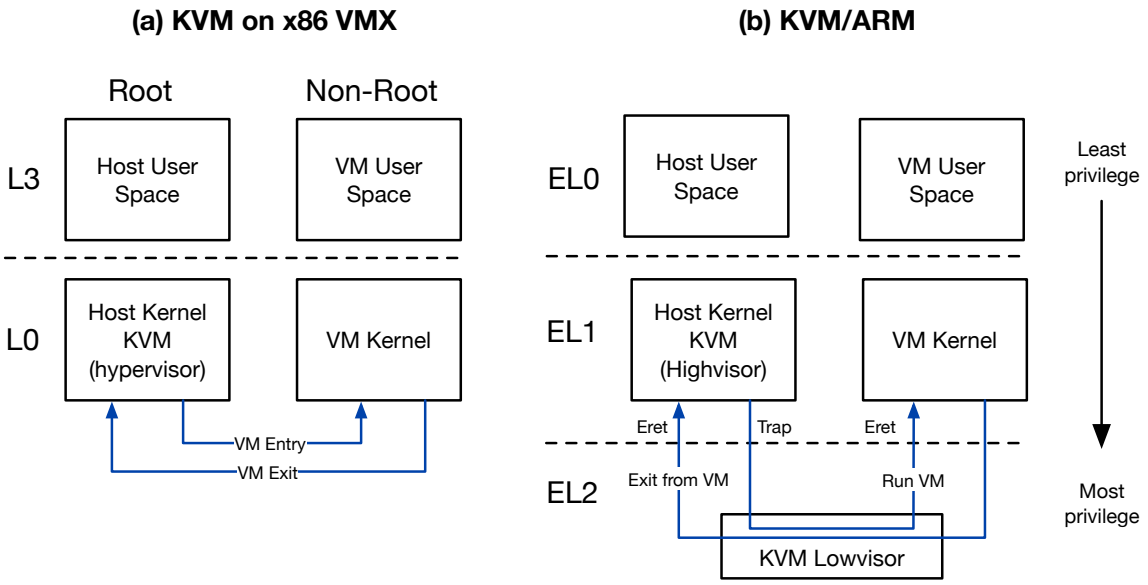

Figure 4.1: KVM on Intel VMX and ARM VE

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 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 4.4.1, 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.

4.2 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 4.1, 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.

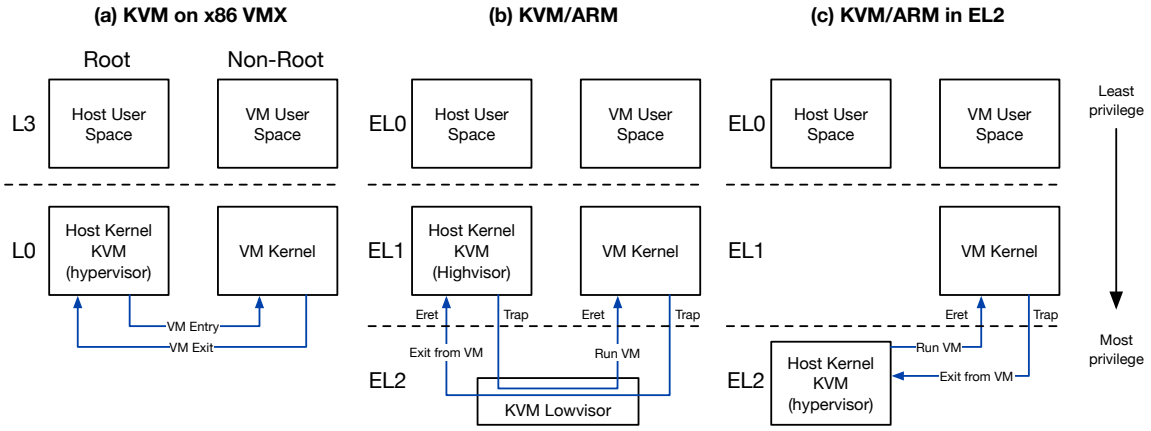

Figure 4.2: Hypervisor Designs and CPU Privilege Levels

Figure 4.2(c) shows how KVM/ARM can be re-designed to run both the hypervisor (KVM) and its hypervisor OS kernel (Linux) together in EL2. Figures 4.2(a) and 4.2(b) are repetitions of

Figures 4.1(a) and 4.1(b), respectively, and shown here again to illustrate the contrast between KVM on x86 (Figure 4.2) which runs the hypervisor OS kernel and the VM kernel in the same CPU mode and privilege level, and KVM/ARM in EL2 (Figure 4.2(c)) which runs the hypervisor OS kernel and the VM kernel in separate CPU modes and privilege levels.

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 [33].

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 incompatibilities 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 4.2(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.

4.2.1 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. ARM based their decision to change their architecture in part based on our work with KVM/ARM presented in Chapter 2 and Chapter 3. VHE

is an architectural hardware modification that provides improved support for hosted hypervisors on ARM.

VHE adds a single bit to the hypervisor control register (HCR_EL2), the E2H bit. VHE maintains complete backwards compatibility with ARMv8.0 and existing hypervisors; if the E2H bit is not set, ARMv8.1 hardware virtualization support behaves exactly the same as ARMv8.0. However, setting the E2H bit significantly changes how the virtualization extensions work and changes the meaning of other control bits, adds new functionality, changes register layouts, and changes the virtual memory system in EL2 [13]. In the following we explain how the three most important features for supporting hypervisor OS kernels running in EL2 work.

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, when enabled, then transparently changes the operation of instructions that normally access EL1 system registers to access EL2 registers instead, when running in EL2. Transparently changing the operation of the instructions lets existing unmodified OSes written to issue EL1 system register instructions instead access EL2 system registers when run in EL2. VHE, when enabled, 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. This is required because a hosted hypervisor typically does not simply run a kernel in isolation, but also runs user space applications to manage VMs. 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. For example, when running a VM, the VM kernel runs in EL1 and VM user space applications run in EL0, and the VM kernel should naturally control the virtual memory configuration of its user space applications. On the other hand, a hypervisor OS kernel running in EL2 will also run user space applications in EL0, and the hypervisor OS kernel should similarly control the virtual memory configuration of the hypervisor OS's user space applications. Therefore, VHE lets the hypervisor OS kernel configure if EL0 is run as part of the hypervisor context, or as part of the VM context, by setting

and clearing the TGE bit, respectively. Configuring EL0 to belong to the hypervisor context by setting the TGE bit also routes exceptions from EL0 directly to EL2, instead of going to EL1. On ARMv8.0, the TGE bit was designed to work with standalone hypervisors only, and it was assumed that a standalone hypervisor's user space applications would be simple bare-metal applications, and the TGE bit therefore simply disabled stage 1 virtual memory in EL0, when EL0 was configured to run as part of the hypervisor context. VHE extends the functionality of the TGE bit such that when VHE is enabled by setting the E2H bit, and exceptions from EL0 are routed to EL2 by setting the TGE bit, virtual memory support is enabled in EL0 and controlled using EL2 page table registers. A hosted hypervisor will typically configure EL0 to use the EL2 system registers when running the hypervisor OS, and configure EL0 to use the EL1 system registers when running the VM.

Third, VHE changes the virtual memory subsystem, known as the *translation regime*, used by EL2 and by EL0 when EL0 is configured to run in the hypervisor context, so that it works exactly in the same way that the EL0/EL1 translation regime works for a kernel running in EL1. There are two main limitations of the EL2 translation regime compared to the EL0/EL1 translation regime before VHE; EL2 uses a different page table format and EL2 only supports a single virtual address range where EL1 supports two, one for the kernel and one for user space. Having two separate address space ranges is convenient and efficient when writing OS software, because one range can be configured for the kernel and be shared across all processes, and the other range can be used for individual user space applications. Only the range used for user space applications need to be reconfigured when switching from one process to the other, and there is no need to maintain mappings of the kernel in every process page table. 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, and VHE also adds the same support known from EL1 to EL2 for two separate virtual address space ranges.

ARMv8.1 differs from the x86 approach in two key ways. First, ARMv8.1 introduces additional hardware register 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 it uses separate register state from EL1. In contrast, recall that x86 adds CPU virtualization support by adding root and non-root operation as orthogonal concepts from the CPU privilege modes, and adds a hardware mechanism to transition between the operations which saves and restores register to memory, but x86 does

not introduce additional hardware register state like ARM. As a result, switching between root and non-root operation requires transferring state between CPU 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 more expensive than having extra hardware register state and not having to perform any memory accesses. 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.

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.

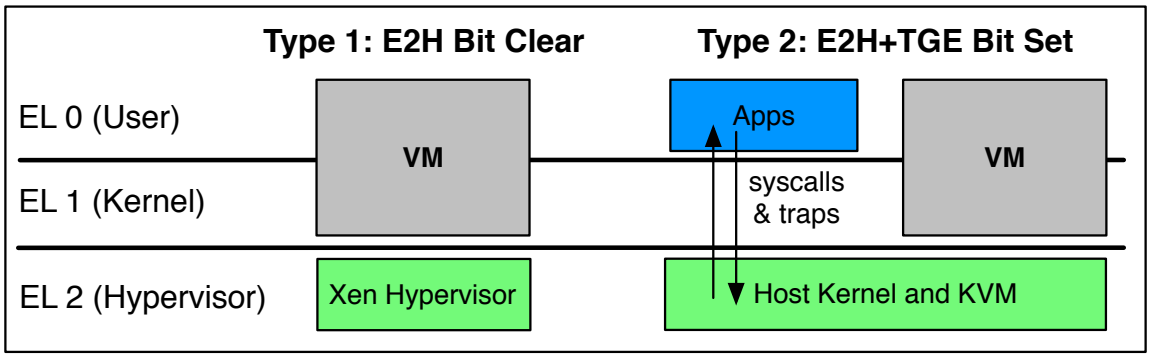

Figure 4.3: Virtualization Host Extensions (VHE)

Figure 4.3 shows how standalone and hosted hypervisors run with VHE. Standalone hypervisors do not set the E2H bit introduced with VHE, and EL2 behaves exactly as in ARMv8 and described in Section 3.1. Hosted hypervisors set the E2H bit when the system boots, and the hypervisor OS kernel runs exclusively in EL2 with both the E2H and TGE bits set, and never runs in EL1.

Hosted hypervisor OS kernels 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 user space to host kernel happen directly from EL0 to EL2, for example to handle a system call, as indicated by the arrows in Figure 4.3. Enabling direct transitions between EL0 and EL2 for the hypervisor host OS no longer disables stage 1 virtual address translations in EL0.

4.2.2 *el2*Linux

Unfortunately, VHE hardware is not yet publicly available and remains an optional extension to the ARM architecture. As an alternative, we introduce KVM/ARM [32], a lightly modified version of Linux that runs in EL2 on non-VHE hardware. KVM/ARM 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 4.4 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. KVM/ARM 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.

KVM/ARM 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.

KVM/ARM 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 sometimes 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, KVM/ARM 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, KVM/ARM 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, KVM/ARM 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.

4.3 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 4.4. 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 under-

lying 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.

4.4 Experimental Results

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 KVM/ARM. We expect that KVM/ARM 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. This is a different machine than used for the performance study presented in Chapter 3, because this machine is equipped with an IOMMU and allows device passthrough, which is the I/O configuration with the best performance and which is the most sensitive to additional latency introduced by the hypervisor architecture. 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 [63] 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 hard-

ware 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. We used a newer version of the Linux kernel than in the performance study presented in Chapter 3, because we wanted to measure what real-world users would realistically run their workloads on when using distribution or mainline Linux kernels, and because we wanted to include optimizations already merged for KVM/ARM between Linux v4.5 and v4.8 and include any optimizations done for KVM x86.

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 implementation changes were merged into Linux. ARM EL2 uses the same KVM/ARM in Linux v4.5 but with modifications to run KVM/ARM 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.

4.4.1 Microbenchmark Results

We first ran various microbenchmarks as listed in Table 4.1, which are part of the KVM unit test framework [53]. 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 4.2 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 ARM results are similar to those shown in Table 3.2 although they vary slightly because a different version of the Linux kernel is used as well as a different CPU microarchitecture. We note that in particular the base VM-to-hypervisor transition cost is very similar, and only the virtual IPI cost is somewhat higher on the platform used for these measurements. The x86 results shown in Table 4.2 are slightly better than those presented in Table 3.2 because of optimizations introduced in KVM in the newer version of the Linux kernel used for the measurements presented in this chapter.

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 performance 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

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 4.1: Microbenchmarks

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 4.2: Microbenchmark Measurements (cycle counts)

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.

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 KVM/ARM 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 4.2.2. 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 4.2. Therefore, most hypervisor configurations do this very rarely. For example, the vhost configuration of virtio [70] 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 [37] 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 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.

4.4.2 Application Benchmark Results

We next ran a mix of widely-used CPU and I/O intensive application workloads as listed in Table 4.3. 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. We chose a subset of the workloads analyzed in the performance study presented in Chapter 3, specifically we excluded SPECjvm2008 and MySQL, because SPECjvm2008 showed almost no overhead, and MySQL results were heavily influenced by changes introduced in Linux between v4.5 and v4.8, unrelated to our work. Figure 4.4 shows the relative performance overhead of executing in a VM compared to natively without virtualization. The ARM and x86 re-

Name	Description
Kernbench	Compilation of the Linux 3.17.0 kernel using the allnoconfig for ARM using GCC 4.8.2.
Hackbench	`hackbench` [62] using Unix domain sockets and 100 process groups running with 500 loops.
Netperf	`netperf` v2.6.0 [52] 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` [78] 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 4.3: Application Benchmarks

sults have different performance characteristics compared to those presented in Chapter 3, because they are run on a different hardware platform using device passthrough instead of paravirtualized I/O, and because they run on a different version of the Linux kernel.

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.

Similar to the microbenchmark measurements in Section 4.4.1, 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

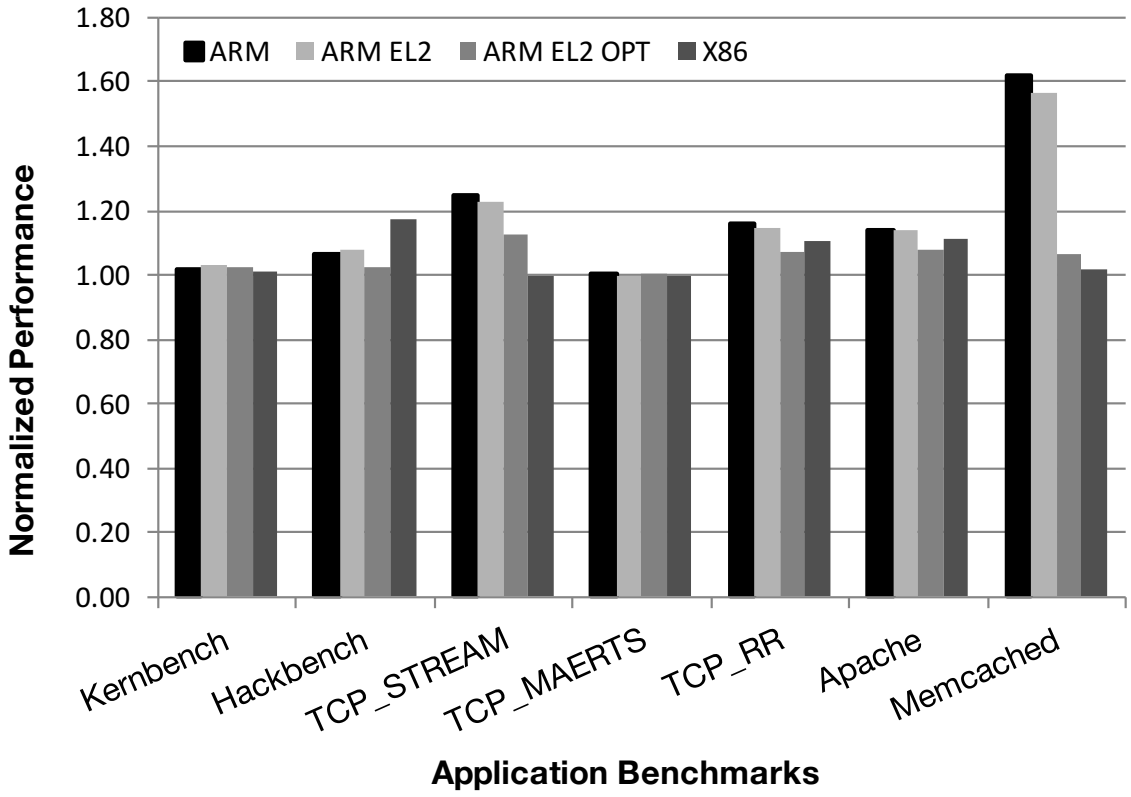

Figure 4.4: Application Benchmark Performance

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 differences, 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.

4.5 Related Work

Agesen et al. [3] show that x86 hardware-based virtualization performance can be improved by using software techniques to coalesce exits from the VM to the hypervisor, even with late generation optimized x86 hardware virtualization support. They were able to leverage significant existing work in a binary translator to rewrite sequences of instructions executed by the guest OS which caused several exits into code that causes only a single exit. Our performance study presented in Chapter 3 did not identify any particular code sequences or repeatable patterns of exits, which could immediately benefit from being rewritten at run time. In fact, our findings indicate that exits from the VM to the hypervisor are very infrequent for CPU and memory-bound workloads and that exits mostly occur when doing I/O.

ELI [41] improves performance of x86 I/O virtualization by directly injecting interrupts from passthrough devices to VMs. When hypervisors use device passthrough (also known as direct device assignment), a single I/O device is assigned to a VM, which is allowed to access the device directly without the hypervisor's intervention. However, there is usually significant overhead from processing interrupts for such a device, because when the device generates interrupts, it causes an exit from the VM to the hypervisor, which must handle the physical interrupt, and emulate injecting a corresponding virtual interrupt to the guest OS. ELI avoids this overhead by leveraging specifics of x86 and creates a shadow interrupt descriptor table which instructs the hardware to jump directly to the guest OS interrupt handler for interrupts from directly assigned devices, and otherwise trap to the hypervisor for other types of interrupts. ARM does not have a similar concept of an interrupt descriptor table, and all interrupts are handled by the same exception vector by the CPU, and the ELI technique therefore doesn't easily apply to ARM. Both x86 and ARM have since received hardware support, posted interrupts and GICv4, respectively, for directly delivering virtual interrupts for assigned devices to guests, but unfortunately no GICv4 hardware is publicly available at the time of writing.

vIC [4] coalesces virtual interrupts for SCSI controllers to reduce the exit rate from VMs to the hypervisor when doing heavy I/O. Similar optimizations may further improve the performance of KVM/ARM, but their optimizations are completely orthogonal to the work presented in this chapter.

Much other work has been also done on analyzing and improving the performance of x86 vir-

tualization [77, 72, 3, 41, 27, 40, 45, 23], but none of these techniques addressed the core issue of the cost of sharing kernel mode across guest and hypervisor OS kernels.

4.6 Summary

In this chapter we discussed the use of hypervisor OS kernels to support the functionality of modern hypervisors. We discussed how the x86 and ARM architectural support for virtualization support hosted hypervisors and we presented a case study of how KVM uses this architectural support to support its Linux hypervisor OS kernel. We presented two ways to improve the support for hypervisor OS kernels, the ARMv8.1 VHE hardware extensions for future hardware, and *el2*Linux, a software solution that works on hardware available at the time of writing. *el2*Linux modifies Linux to run in the ARM hypervisor CPU mode, EL2. We then presented a new hypervisor design, which takes advantage of running the entire hypervisor including its hypervisor OS kernel in a separate CPU mode from the VM kernel. Our results show an order of magnitude reduction in overhead for the base cost of trapping from the VM to the hypervisor OS kernel, measured by the hypercall microbenchmark, and we show more than 50% reduction in overhead for important application benchmark results.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

This dissertation explores new approaches to combining software and architectural support for virtualization with a focus on the ARM architecture and shows that it is possible to provide virtualization services an order of magnitude more efficiently than traditional implementations.

Chapter 1 investigated limitations to virtualizing the ARM instruction set without architectural support for virtualization, and we identified 28 instructions on ARMv7 which are sensitive and non-privileged. We then presented an early prototype of KVM for ARM that uses lightweight paravirtualization to address the problem with sensitive non-privileged instructions on ARM. Lightweight paravirtualization is a simpler approach compared to full paravirtualization. Full paravirtualization replaces entire functions in the guest OS by hand, and requires ongoing maintenance of every supported guest OS. Lightweight paravirtualization is fully automated and statically replaces individual sensitive non-privileged instructions with privileged instructions that trap to the hypervisor. The early prototype of KVM for ARM defines an encoding of all sensitive non-privileged instructions including their operands into privileged instructions, allowing sensitive instructions to be replaced one-by-one and leaving the rest of the guest OS code intact.

Chapter 2 presented split-mode virtualization, a new hypervisor design that splits the core hypervisor across different CPU modes and allows a single code base to take advantage of the features offered from both CPU modes. Split-mode virtualization is used to support hosted hypervisor designs using the ARM virtualization extensions without having to modify the host OS. Hosted virtualization has key benefits in the context of ARM virtualization, where systems are often implemented without following any standards, often have custom boot protocols instead of relying on a well-defined interface to BIOS or UEFI firmware, and often times do not have any discoverable bus such as PCI. On such systems, software cannot rely on standards or legacy memory maps, and as a

results standalone hypervisors have to be ported to every single supported platform. Hosted hypervisors, on the other hand, can reuse the device and platform support of the host OS. KVM/ARM, using split-mode virtualization, was designed from the ground up to be included in mainline Linux so that it would be widely used in practice and could serve as a foundation for future research, as well as for production use. KVM/ARM for ARMv7 was included in mainline Linux v3.9 and 64-bit ARMv8 support followed in Linux v3.10. As a result, KVM/ARM is supported on almost all systems that support Linux and are equipped with the ARM virtualization extensions without further porting efforts. Based on my experiences getting code merged into the Linux kernel, and subsequently maintaining the code for several years while seeing it widely adopted for production use, we offer advice to others in how to get research ideas adopted by open source communities.

Chapter 3 presented an in-depth study of ARM virtualization performance using the two main ARM hypervisors, KVM/ARM and Xen for ARM, on 64-bit ARMv8 server hardware. We provided x86 measurements using both KVM and Xen as a baseline for comparison. Given that the ARM virtualization extensions were initially designed for standalone hypervisors, we carefully analyzed the performance implications of using split-mode virtualization and compared both microbenchmark results and application benchmark results using both KVM/ARM, a hosted hypervisor, and Xen, a standalone hypervisor. Our study shows that ARM enables standalone hypervisors such as Xen to transition between the VM and the hypervisor three times faster than on x86, but this low transition cost does not extend to hosted hypervisors such as KVM/ARM which is five times slower on ARM than on x86 for the same hypervisor design. Standalone hypervisors on ARM are able to achieve such good results, because the standalone hypervisor runs in a CPU mode separate from the VM, and there is no need to multiplex any register state between the hypervisor and VM execution contexts. However, these performance characteristics are not well correlated with application performance. For real application workloads involving I/O, the hypervisor's I/O model dwarfs the difference in cost when switching between the VM and the hypervisor. Xen runs a special privileged VM, Dom0, to perform I/O on behalf of the application VM, and Xen has to facilitate communication between I/O backends in Dom0 and the application VM, requiring much more complex interactions than simply transitioning to and from the hypervisor CPU mode. A key conclusion from our study is that both hypervisors were unable to leverage ARM's potential for fast transitions between the hypervisor and the VM for real application workloads. In fact, KVM/ARM using split-mode virtualization

performs better than Xen on ARM for real application workloads using I/O, because KVM/ARM benefits from being integrated directly with an OS kernel and does not rely on running an additional VM, such as Dom0, to run I/O on behalf of other VMs and because KVM/ARM's I/O model allows full access from the hypervisor to application VMs which facilitates zero copy I/O and significantly improves performance.

Chapter 4 presented a new hypervisor design for KVM/ARM which takes advantage of running both the hypervisor and its hypervisor OS kernel in a separate CPU mode from the VM. Our new hypervisor design benefits from new architectural features introduced in ARMv8.1, the Virtualization Host Extensions (VHE), to avoid modifications to the hypervisor OS kernel. We introduced *el2*Linux, a port of Linux that runs in ARM's EL2 hypervisor mode to run the hypervisor and its OS kernel in the same CPU mode on legacy systems without VHE. Our new hypervisor design improves real application performance and reduces virtualization overhead by more than 50%, and achieves very low VM-to-hypervisor transition costs, lower than on x86 and an order of magnitude lower than previous designs. We achieve this by combining software and architectural support for virtualization in a new way which avoids saving and restoring any CPU state when switching between the VM and the hypervisor, but instead runs the hypervisor and its hypervisor OS kernel in a dedicated CPU mode with its own register state separate from the VM's.

Building the first and most popular industry ready hypervisor for a computer architecture using new architectural features is a rare, unique, and multi-faceted challenge, from which important and interesting lessons can be extracted. Much of the work on ARM virtualization presented in this dissertation coincided with ARM being deployed as servers and networking equipment [64, 73], in addition to the mobile and embedded markets where ARM remains the dominating architecture. This change in the world's computing ecosystem allowed me to apply research ideas in practice and evaluate techniques both academically and in practice.

We have, for the first time, the benefit of being able to compare virtualization software and architectural support for virtualization on two widely deployed architectures. x86 architectural support for virtualization supports running full operating systems in both its root and non-root operation but requires context switching the CPU state between the two operations. ARM, on the other hand, provides a separate CPU mode to run the hypervisor and avoids the need to context switch the CPU state between the hypervisor CPU mode and other modes. Why did these two architectures choose

such fundamentally different designs? It was natural for ARM to design their architectural support around existing features in the ISA and expand the protection mechanism with an additional CPU mode because it follows the design principle of RISC architectures and introduces few and simple changes to the ARM ISA. The ARM virtualization extensions provide the bare minimum feature set for running a hypervisor, leaving complicated operations to software. In contrast, x86 is a CISC architecture and already has a complicated ISA, and it was natural to simply introduce the expressive power required to run VMs directly in the ISA.

An initial drawback of the ARM design compared to the x86 design was the lack of support for running full OS kernels in the hypervisor CPU mode. This design decision was the unfortunate result of trying to simplify the architecture and neglecting an existing software design; hosted hypervisors. However, with the introduction of VHE this oversight was addressed and in some way the architectural support for virtualization in ARM and x86 are now equally capable — but they have very different performance characteristics. ARM ends up being able to provide significantly lower VM-to-hypervisor transition costs compared to x86 for both standalone and hosted hypervisors because ARM implementations introduce additional hardware register state for the dedicated hypervisor CPU mode and avoids saving and restoring state to memory. With VHE, ARM supports running the full hypervisor and its hypervisor OS kernel in the dedicated hypervisor CPU mode and the performance benefit of the ARM virtualization extensions extend to real application performance as well. The key lesson is that when designing architectural support for virtualization, the architecture should support separate hypervisor and VM execution contexts where both contexts support running a full OS kernel, and to achieve good performance the architecture should facilitate very fast switching between the hypervisor and the VM execution contexts.

As a final contribution, I wish to offer my opinion on an ongoing discussion about the design of hypervisor software. Are standalone (Type 1) or hosted (Type 2) hypervisors better? I take no definitive stance in this debate. Each design has its merits depending on the deployment scenario and the surrounding software ecosystem. For example, VMware ESXi is a commercial standalone hypervisor and it can limit its base of supported hardware to select server systems and include drivers for supported hardware within the hypervisor without relying on an existing OS and still provide fast I/O virtualization compared to other standalone hypervisors such as Xen which runs I/O drivers in a VM. Microsoft Hyper-V, also a standalone hypervisor, pays the cost of running

the device drivers in a VM, possibly because it reduces the hypervisor TCB and because security and isolation are important factors for Microsoft's customer base. On the other hand, the core idea of KVM, a hosted hypervisor, is to reuse existing Linux features and share code maintenance and platform support with the Linux kernel and let users, who already run Linux, easily run VMs using familiar tools and configurations. VMware workstation, another hosted hypervisor, is designed to be able to run on any developer laptop and therefore has to rely on existing drivers in the host OS. Furthermore, WMware workstation is not amenable to a standalone design but instead installs itself as a module in a host OS because of its distribution model as a downloadable and installable application.

The important observation is that both hypervisor designs are facts of life and commonly used, and simply classifying a hypervisor as either hosted or standalone does not provide a full picture of the suitability or performance of the hypervisor. The glaringly obvious example is the case of the ARM virtualization extensions where it was originally assumed that standalone hypervisors would be more suitable and have better performance because the initial ARM architectural support for virtualization was designed specifically for standalone hypervisors; however, it was KVM/ARM, a hosted hypervisor, which became the first and most popular ARM hypervisor to be widely deployed in production due to the significant benefits of being integrated with Linux in the ARM ecosystem.

5.2 Future Work

My research has investigated core aspects of virtualization on ARM, especially related to architectural support for virtualization, and the design, implementation, and evaluation of hypervisors on ARM. There is, however, some virtualization techniques which have been explored on x86 and not yet explored on ARM, as well as a number of deployment scenarios which more commonly use ARM processors, such as mobile phones and embedded computing, where virtualization may also be useful.

Nested virtualization [18], also known as recursive virtualization, is the discipline of running VMs within VMs. Since this work is closely tied to the architectural support for virtualization, it should be explored how to support nested virtualization on ARM. One immediate problem is that the ARM virtualization extensions are not self-virtualizable because EL2 instructions do not trap

when executed in EL1, and therefore EL2 cannot be emulated in a less privileged CPU mode using trap-and-emulate. ARM has recently announced changes in the ARMv8.3 architecture version to support nested virtualization [21], and we have proposed further extensions to the architecture [57] which are included in the ARMv8.4 architecture version to improve the performance of nested virtualization on ARM. Nested virtualization is an increasingly popular technology for supporting guest OSes with built-in virtualization functionality in the cloud and is heavily used for test, development, and continuous integration. However, even with the upcoming changes in ARMv8.4, preliminary results show that paravirtualized I/O performance in nested VMs can be more than 5X slower compared to native performance. Direct device assignment beyond one level of virtualization requires multiple levels of translations for DMA transactions and requires emulating virtual IOMMUs to each level of guest hypervisors. Preliminary results show that also this approach to providing I/O virtualization in the context of nested virtualization causes high overhead from the cost of having to trap and emulate multiple levels of DMA remapping operations on the virtual IOMMUs and propagating these mappings to the physical IOMMU. For these reasons, further research in how to provide sufficiently efficient virtual I/O for multiple levels of virtualization is important.

As requirements for compute bandwidth and throughput continues to increase, for example for computer vision and machine learning workloads, the use of custom accelerators and FPGAs is on the rise. Recently announced system interconnects such as CCIX, Gen-Z, and OpenCAPI indicate a trend towards more heterogeneous system designs consisting of a mix of compute nodes with potentially different ISAs, coupled with accelerators and FPGAs in cache-coherent interconnects. As these workloads move away from custom installations towards general-purpose cloud computing, it will inevitably be necessary to support VMs on heterogeneous systems which have direct access to accelerators, FPGAs, and even support running across CPU cores with different ISAs. Current device assignment support for VMs on both ARM and x86 is mostly limited to PCIe devices, and it is unclear how to multiplex and assign resources which are more tightly integrated using new system interconnects.

As cloud computing becomes more and more pervasive, and we move an increasing amount of private and sensitive data to the cloud, security and reliability of cloud computing is becoming a key concern for both cloud providers and society. ARM's virtualization extensions provide a separate CPU mode to run the hypervisor, and the split-mode virtualization design presented in Chapter 2

runs a very small amount of code in the most privileged CPU mode. This smaller TCB can potentially be leveraged to provide better security guarantees for virtualization solutions than traditional hypervisor solutions which relies on a much larger TCB. ARM's design of architectural support for virtualization offers unique benefits in being able to quickly switch between CPU modes and potentially allows running a small TCB in the most privileged CPU mode to ensure the integrity of workloads running in less privileged CPU modes, even outside the context of virtualization. For example, stage 2 memory translation could be enabled with relatively low overhead to protect certain memory regions from unwanted access or to validate critical accesses against system integrity measures. It is also of key importance to explore how computer architectures could be changed to provide increased security and isolation for cloud deployments and which effects that may have on software.

Another way of improving the reliability and security of virtualization is by formally verifying correct functionality of the hypervisor. This is relevant not only for data center and cloud infrastructure, but also for embedded and safety-critical deployments which are moving towards more powerful and centralized compute nodes, using virtualization instead of hardware separation, to provide isolation between workloads. Existing work has formally verified an abstract specification of a microkernel OS [55], but this OS is not deployed in practice for many applications of virtualization. As discussed in Chapters 2 and 4, code maintainability and reuse of existing driver and hardware support is key for many practical deployments. Applying formal verification techniques to hypervisors which build on existing large software code bases to improve their reliability and reduce vulnerabilities would make a real-world impact and should be explored in more depth.

Another area where ARM is becoming increasingly commonplace is automotive. Modern cars typically have several hundred integrated control units, many of them are based on small ARM microprocessors. There is a general interest in consolidating these tiny nodes into larger compute cores to reduce cost and complexity of automotive systems, and virtualization has been suggested as one way to maintain isolation between various critical systems. However, there are many challenges in using virtualization technology in automotive and safety critical systems in general. For example, real-time sensitive tasks may not meet their real-time requirements in VMs, the isolation guarantees provided by virtualization may not be strong enough compared to running on physically separate systems, and there are reliability concerns because many systems could fail if a single hardware unit

malfunctions. Understanding the benefits and risk in bringing virtualization to automotive systems is an immensely important challenge.

The ARM architecture is making a push into the server and networking markets, but also remain dominating for smartphones and tablets. Previous work have explored various techniques to support multiple execution environments on these types of devices using operating system virtualization and binary compatibility to solve problems otherwise commonly solved with virtualization on x86-based PCs. Cells [6] uses operating system virtualization to create lightweight containers supporting multiple Android instances on a smartphone. Cider [7, 5] uses binary compatibility to run iOS applications side-by-side with Android applications using the same operating system kernel. Both of these use cases could also be solved using full system virtualization, for example building on the KVM/ARM work presented in this dissertation, but there are unsolved challenges in doing so. To run full Android instances with virtually unrestricted features and user interactions, one must solve the problem of multiplexing peripherals like the GPU, touchscreen, phone subsystem, sensor devices, and audio. Running a proprietary operating system like iOS in a VM would require emulating the devices expected by the guest OS or would require being able to configure the guest OS for the virtual emulated platform by for example installing custom drivers, similar to what users do with Microsoft Windows on PCs. In both cases, understanding the performance and latency characteristics of full system VMs on mobile hardware platforms would be essential in understanding the feasibility of virtualization as a technology to solve use cases requiring multiple execution environments on mobile. A key challenge in providing any user-facing virtualization experience is GPU virtualization, which has only been partially solved for virtual desktop infrastructure (VDI) deployments. Exploring how hardware and software can be co-designed to make it easier to provide GPU virtualization across a wide range of hypervisor and guest OS implementation, could make virtualization a more useful technology for end-user facing deployments such as tablets and mobile phones.

Finally, the ARM architecture features the Security Extensions, also known as TrustZone. TrustZone provides separate secure CPU privilege modes designed to run Trusted Execution Environments (TEEs) underneath a full system stack consisting of potentially a hypervisor, OS, and user space applications. Currently, running a virtual machine that supports the full range of ARM architecture features including TrustZone is only supported using pure software emulation, which is

orders of magnitude slower than virtualization. There are interesting use cases for emulating TrustZone in VMs, for example to support proprietary system images that are designed to be installed and booted in the secure CPU modes on ARM, or for test and development, which is currently cumbersome and slow and requires complicated certificate configurations on real hardware or using slow software emulators. It is potentially possible to emulate both secure and non-secure environments for a single VM using ARM's virtualization extensions by creating isolated environments for the secure and non-secure world, and by emulating the behavior of the EL3 monitor either in software or using trap-and-emulate. Investigating the possibilities of this support would close the loop in understanding ARM's capability to be fully self-virtualizing.

Bibliography

[1] Keith Adams and Ole Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '06, pages 2–13, October 2006.

[2] Advanced Micro Devices. AMD64 Architecture Programmers Manual, Volume 2: System Programming rev. 3.23, May 2013.

[3] Ole Agesen, Jim Mattson, Radu Rugina, and Jeffrey Sheldon. Software Techniques for Avoiding Hardware Virtualization Exits. In *Proceedings of the 2012 USENIX Annual Technical Conference*, USENIX ATC '12, pages 373–385, June 2012.

[4] Irfan Ahmad, Ajay Gulati, and Ali Mashtizadeh. vIC: Interrupt Coalescing for Virtual Machine Storage Device IO. In *Proceedings of the 2011 USENIX Annual Technical Conference*, USENIX ATC '11, pages 45–58, June 2011.

[5] Jeremy Andrus, Naser AlDuaij, and Jason Nieh. Binary Compatible Graphics Support in Android for Running iOS Apps. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, Middleware '17, pages 55–67, 2017.

[6] Jeremy Andrus, Christoffer Dall, Alexander Van't Hof, Oren Laadan, and Jason Nieh. Cells: A Virtual Mobile Smartphone Architecture. In *Proceedings of the 23rd Symposium on Operating Systems Principles*, SOSP '11, pages 173–187, October 2011.

[7] Jeremy Andrus, Alexander Van't Hof, Naser AlDuaij, Christoffer Dall, Nicolas Viennot, and Jason Nieh. Cider: Native execution of iOS apps on Android. In *Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 367–382, March 2014.

[8] ARM Ltd. ARM Energy Probe. http://www.arm.com/products/tools/arm-energy-probe.php.

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

[10] ARM Ltd. ARM Cortex-A15 Technical Reference Manual ARM DDI 0438C, September 2011.

[11] ARM Ltd. ARM Architecture Reference Manual ARMv7-A DDI0406C.c, May 2014.

[12] ARM Ltd. ARM Generic Interrupt Controller Architecture version 3.0 and version 4.0 ARM IHI 0069C, July 2016.

[13] ARM Ltd. ARM Architecture Reference Manual ARMv8-A DDI0487B.a, March 2017.

[14] ARM Ltd. ARMv7-M Architecture Reference Manual ARMv7-M ARM DDI 0403E.c, May 2017.

[15] ARM Ltd. ARMv8-M Architecture Reference Manual ARMv8-M DDI 0553A.e, June 2017.

[16] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, October 2003.

[17] Ken Barr, Prashanth Bungale, Stephen Deasy, Viktor Gyuris, Perry Hung, Craig Newell, Harvey Tuch, and Bruno Zoppis. The VMware Mobile Virtualization Platform: is that a hypervisor in your pocket? *SIGOPS Operating Systems Review*, 44(4):124–135, December 2010.

[18] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, 2010.

[19] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating Two-Dimensional Page Walks for Virtualized Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '08, pages 26–35, March 2008.

[20] Paolo Bonzini. Virtualizing the locomotive, September 2005. https://lwn.net/Articles/657282/.

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

[22] David Brash. Extensions to the ARMv7-A Architecture. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, August 2010.

[23] Jeffrey Buell, Daniel Hecht, Jin Heo, Kalyan Saladi, and H. Reza Taheri. Methodology for Performance Analysis of VMware vSphere under Tier-1 Applications. *VMware Technical Journal*, 2(1):19–28, June 2013.

[24] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y. Wang. Bringing Virtualization to the x86 Architecture with the Original VMware Workstation. *ACM Transactions on Computer Systems*, 30(4):12:1–12:51, November 2012.

[25] Edouard Bugnion, Jason Nieh, and Dan Tsafrir. *Hardware and Software Support for Virtualization*, volume 12 of *Synthesis Lectures on Computer Architecture*. Morgan & Claypool Publishers, February 2017.

[26] Ian Campbell. Xen ARM with Virtualization Extensions. http://wiki.xen.org/wiki/Xen_ARM_with_Virtualization_Extensions.

[27] 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*, USENIX ATC '05, pages 387–390, June 2005.

[28] Jonathan Corbet. Memory management notifiers. LWN.net, January 2008. https://lwn.net/Articles/266320.

[29] Jonathan Corbet. Rationalizing the arm tree. LWN.net, April 2011. https://lwn.net/Articles/439314/.

[30] Christoffer Dall, Jeremy Andrus, Alexander Van't Hof, Oren Laadan, and Jason Nieh. The Design, Implementation, and Evaluation of Cells: A Virtual Smartphone Architecture. *ACM Transactions on Computer Systems*, 30(3):9:1–9:31, August 2012.

[31] Christoffer Dall and Andrew Jones. KVM/ARM Unit Tests. https://github.com/columbia/kvm-unit-tests.

[32] Christoffer Dall and Shih-Wei Li. *el2*linux. https://github.com/chazy/el2linux.

[33] Christoffer Dall, Shih-Wei Li, Jin Tack Lim, Jason Nieh, and Georgios Koloventzos. ARM virtualization: Performance and architectural implications. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture*, ISCA '16, pages 304–316, June 2016.

[34] Christoffer Dall and Jason Nieh. KVM for ARM. In *Proceedings of the Ottawa Linux Symposium (OLS)*, pages 45–56, July 2010.

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

[36] Christoffer Dall and Jason Nieh. Supporting KVM on the ARM architecture. LWN.net, July 2013. http://lwn.net/Articles/557132/.

[37] Christoffer Dall and Jason Nieh. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 333–348, March 2014.

[38] David Brash, Architecture Program Manager, ARM Ltd. Personal communication, November 2012.

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

[40] Kangarlou Gamage and Xu Kompella. Opportunistic Flooding to Improve TCP Transmit Performance in Virtualized Clouds. In *Proceedings of the 2nd Symposium on Cloud Computing*, SOCC '11, pages 24:1–24:14, 2011.

[41] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. ELI: Bare-metal performance for I/O virtualization. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 411–422, March 2012.

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

[43] The Tcpdump Group. Tcpdump. http://www.tcpdump.org/tcpdump_man.html.

[44] Gernot Heiser and Ben Leslie. The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors. In *Proceedings of the 1st ACM SIGCOMM Asia-Pacific Workshop on Systems*, ApSys 2010, pages 19–24, August 2010.

[45] Jin Heo and Reza Taheri. Virtualizing Latency-Sensitive Applications: Where Does the Overhead Come From? *VMware Technical Journal*, 2(2):21–30, December 2013.

[46] Hewlett-Packard. HP Moonshot-45XGc switch module. http://www8.hp.com/us/en/products/moonshot-systems/product-detail.html?oid=7398915, Accessed: Feb. 2016.

[47] Joo-Young Hwang, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. Xen on ARM: System Virtualization using Xen Hypervisor for ARM-based Secure Mobile Phones. In *Proceedings of the 5th Consumer Communications and Newtork Conference*, January 2008.

[48] Ian Campbell. Personal communication, April 2015.

[49] InSignal Co. ArndaleBoard.org. http://arndaleboard.org.

[50] Intel Corporation. Intel 64 and IA-32 Architectures Software Developers Manual, 325384-058US, April 2016.

[51] Intel Corporation. Intel Virtualization Technology for Directed I/O, D51397-008, Rev.2.4, June 2016.

[52] Rick Jones. Netperf. http://www.netperf.org/netperf.

[53] Avi Kivity. KVM Unit Tests. https://git.kernel.org/cgit/virt/kvm/kvm-unit-tests.git.

[54] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. **kvm**: The Linux Virtual Machine Monitor. In *Proceedings of the Ottawa Linux Symposium (OLS)*, volume 1, pages 225–230, June 2007.

[55] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, 2009.

[56] KVM/ARM Mailing List. https://lists.cs.columbia.edu/cucslists/listinfo/kvmarm.

[57] Jin Tack Lim, Christoffer Dall, Shih-Wei Li, Jason Nieh, and Marc Zyngier. NEVE: Nested Virtualization Extensions for ARM. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 201–217, 2017.

[58] Linux ARM Kernel Mailing List. A15 H/W Virtualization Support, April 2011. http://archive.arm.linux.org.uk/lurker/message/20110412.204714.a36702d9.en.html.

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

[60] Paul McKenney. Arm kernel consolidation. LWN.net, May 2011. https://lwn.net/Articles/443510/.

[61] Larry McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. In *Proceedings of the 1996 USENIX Annual Technical Conference*, USENIX ATC '96, pages 279–294, January 1996.

[62] Ingo Molnar. Hackbench. http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c.

[63] Univeristy of Utah. CloudLab. http://www.cloudlab.us.

[64] Packet. ARMv8 in the Datacenter. https://www.packet.net/bare-metal/servers/type-2a/. Accessed: Jun. 2017.

[65] Sandro Pinto, Jorge Pereira, Tiago Gomes, Mongkol Ekpanyapong, and Adriano Tavares. Towards a trustzone-assisted hypervisor for real time embedded systems. *IEEE Computer Architecture Letters*, PP(99), October 2016.

[66] Gerald J. Popek and Robert P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM*, 17:412–421, July 1974.

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

[68] Richard Grisenthwaite, Lead Architect and Fellow, ARM Ltd. Personal communication, April 2015.

[69] Richard Grisenthwaite, Lead Architect and Fellow, ARM Ltd. Personal communication, October 2016.

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

[71] Rusty Russell. Virtio PCI Card Specificaion v0.9.5, May 2012. https://ozlabs.org/~rusty/virtio-spec/virtio-0.9.5.pdf.

[72] Jose Renato Santos, Yoshio Turner, G. John Janakiraman, and Ian Pratt. Bridging the Gap between Software and Hardware Techniques for I/O Virtualization. In *Proceedings of the 2008 USENIX Annual Technical Conference*, USENIX ATC '08, pages 29–42, June 2008.

[73] Scaleway. Scaleway ARMv8 Cloud Servers. https://www.scaleway.com. Accessed: Jun. 2017.

[74] Spec.org. Specjvm2008. https://www.spec.org/jvm2008.

[75] Stefano Stabellini. Tuning Xen for Performance. http://wiki.xen.org/wiki/Tuning_Xen_for_Performance. Accessed: Jul. 2015.

[76] Udo Steinberg and Bernhard Kauer. Nova: A Microhypervisor-Based Secure Virtualization Architecture. In *Proceedings of the 5th European Conference on Computer Systems*, EUROSYS '10, pages 209–222, April 2010.

[77] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference*, USENIX ATC '01, pages 1–14, June 2001.

[78] The Apache Software Foundation. ab, April 2015. http://httpd.apache.org/docs/2.4/programs/ab.html.

[79] Michael S. Tsirkin, Cornelia Huck, and Pawel Moll. Virtual I/O Device (VIRTIO) v1.0, March 2016. http://docs.oasis-open.org/virtio/virtio/v1.0/virtio-v1.0.html.

[80] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. Intel Virtualization Technology. *IEEE Computer*, 38(5):48–56, May 2005.

[81] Prashant Varanasi and Gernot Heiser. Hardware-Supported Virtualization on ARM. In *Proceedings of the Second SIGCOMM Asia-Pacific Workshop on Systems*, ApSYS 2011, pages 11:1–11:5, July 2011.

[82] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. Technical Report 02-02-01, University of Washington, 2002.

[83] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and Performance in the denali Isolation Kernel. In *Proceedings of the 5th symposium on Operating systems design and implementation*, OSDI '02, 2002.

[84] Xenproject.org. CVE-2017-10912. CVE-ID CVE-2017-10912., July 2017. https://xenbits.xen.org/xsa/advisory-217.html. Accessed: Aug. 2017.

[85] Xenproject.org. CVE-2017-10917. CVE-ID CVE-2017-10917., July 2017. https://xenbits.xen.org/xsa/advisory-221.html. Accessed: Aug. 2017.

[86] Xenproject.org. CVE-2017-10918. CVE-ID CVE-2017-10918., July 2017. https://xenbits.xen.org/xsa/advisory-222.html. Accessed: Aug. 2017.

[87] Marc Zyngier. ARM Caches: Giving you enough rope to shoot yourself in the foot. KVM Forum 2015. https://events.linuxfoundation.org/sites/events/files/slides/slides_10.pdf.