KVM/ARM: Experiences Building the Linux ARM Hypervisor

Christoffer Dall and Jason Nieh
{cdall, nieh}@cs.columbia.edu
Department of Computer Science, Columbia University

Technical Report CUCS-010-13
April 2013

Abstract

As ARM CPUs become increasingly common in mobile devices and servers, there is a growing demand for providing the benefits of virtualization for ARM-based devices. We present our experiences building the Linux ARM hypervisor, KVM/ARM, the first full system ARM virtualization solution that can run unmodified guest operating systems on ARM multicore hardware. KVM/ARM introduces split-mode virtualization, allowing a hypervisor to split its execution across CPU modes to take advantage of CPU mode-specific features. This allows KVM/ARM to leverage Linux kernel services and functionality to simplify hypervisor development and maintainability while utilizing recent ARM hardware virtualization extensions to run application workloads in guest operating systems with comparable performance to native execution. KVM/ARM has been successfully merged into the mainline Linux 3.9 kernel, ensuring that it will gain wide adoption as the virtualization platform of choice for ARM. We provide the first measurements on real hardware of a complete hypervisor using ARM hardware virtualization support. Our results demonstrate that KVM/ARM has modest virtualization performance and power costs, and can achieve lower performance and power costs compared to x86-based Linux virtualization on multicore hardware.

1 Introduction

ARM-based devices are seeing tremendous growth across smartphones, netbooks, and embedded computers. While ARM CPUs have benefited from their advantages in power efficiency 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. This is spurring the development of new ARM-based microservers and an upward push of ARM CPUs into traditional server and PC systems.

Unlike x86-based systems, a key limitation of ARM-based systems has been the lack of support for virtualization. To address this problem, ARM has introduced hardware virtualization extensions in the newest ARM CPU architectures. ARM has benefited from the hindsight of x86 in its design. For example, nested page tables, not part of the original x86 virtualization hardware, are standard in ARM. However, there are important differences between ARM and x86 virtualization extensions such that x86 hypervisor designs may not be directly amenable to ARM. These differences may also impact hypervisor performance, especially for multicore systems, but have not been evaluated with real hardware.

We describe our experiences building KVM/ARM, the ARM hypervisor in the mainline Linux kernel. KVM/ARM is the first hypervisor to leverage ARM hardware virtualization support to run unmodified guest operating systems (OSes) on ARM multicore hardware. Our work makes four main contributions. First, we introduce split-mode virtualization, a new approach to hypervisor design that splits the core hypervisor so that it runs across different 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 introduces a new CPU mode for running hypervisors called Hyp mode, but Hyp mode has a different feature set from other CPU modes. Hypervisors provide many aspects of OS functionality, but standard OS mechanisms in Linux would have to be significantly redesigned to run in Hyp mode. Our split-mode virtualization mechanism allows a hypervisor to use Hyp mode to leverage ARM hardware virtualization features, but also run in normal privileged CPU modes, allowing it to coexist with other OS functionality.

Second, we designed and implemented KVM/ARM from the ground up as an open source project that would be easy to maintain and integrate into the Linux kernel. For example, by using our split-mode virtualization, we can leverage the existing KVM hypervisor interface in Linux and can reuse substantial pieces of existing kernel code and interfaces to reduce code duplication. KVM/ARM was accepted as the ARM hypervisor of the mainline Linux kernel as of the Linux 3.9 kernel, ensuring its wide adoption and use given the dominance of Linux on ARM platforms. Based on our open source experiences, we offer some useful hints on transferring research ideas into implementations likely to be adopted by existing open source communities.

Third, we demonstrate the effectiveness of KVM/ARM on real multicore ARM hardware. Our results are the first measurements of a hypervisor using ARM virtualization support on real hardware. We compare against the standard widely-used Linux x86 KVM hypervisor and evaluate its performance overhead for running application workloads in virtual machines (VMs) versus native non-virtualized execution. Our results show that KVM/ARM achieves comparable performance overhead in most cases, and significantly lower performance overhead for two important applications, Apache and MySQL, on multicore platforms. These results provide the first comparison of ARM and x86 virtualization extensions on real hardware to quantitatively demonstrate how the different design choices affect virtualization performance. We show that KVM/ARM also provides power efficiency benefits over Linux x86 KVM.

Finally, we make several recommendations regarding future hardware support for virtualization based on our experiences building and evaluating a complete ARM hypervisor. We identify features that are important and helpful to reduce the software complexity of hypervisor implementation, and discuss mechanisms useful to maximize hypervisor performance, especially in the context of multicore systems.

This technical report describes our experiences designing, implementing, and evaluating KVM/ARM. Section 2 presents an overview of the ARM virtualization extensions and a comparison with x86. Section 3 describes the design of the KVM/ARM hypervisor. Section 4 discusses the implementation of KVM/ARM and our experiences releasing it to the Linux community and having it adopted into the mainline Linux kernel. Section 5 presents experimental results quantifying the performance and energy efficiency of KVM/ARM, as well as a quantitative comparison of real ARM and x86 virtualization hardware. Section 6 makes several recommendations about designing hardware support for virtualization. Section 7 discusses related work. Finally, we present some concluding remarks.

2 ARM Virtualization Extensions

Because the ARM architecture is not classically virtualizable [20], ARM has introduced hardware virtualization support as an optional extension in the latest ARMv7 architecture [4] and a mandatory part of the upcoming 64-bit ARMv8 architecture. The Cortex-A15 [2] is an examples of current ARMv7 CPUs including hardware virtualization extensions. We present a brief overview of the ARM virtualization extensions.

2.1 CPU Virtualization

Figure 1 shows the CPU modes on the ARMv7 architecture, including TrustZone (Security Extensions) and a new CPU mode called Hyp mode. TrustZone splits the modes into two worlds, secure and non-secure, which are orthogonal to the CPU modes. A special mode, monitor mode, is provided to switch between the secure and non-secure worlds. Although ARM CPUs always power up starting in the secure world, ARM bootloaders typically transition to the non-secure world at an early stage and secure world is only used for specialized use cases such as digital rights management. TrustZone may appear useful for virtualization by using the secure world for hypervisor execution, but this does not work because there is no support for trap-and-emulate. There is no means to trap operations executed in the non-secure world to the secure world. Non-secure software can therefore freely configure, for example, virtual memory. Any software running in the non-secure world therefore has access to all non-secure memory, making it impossible to isolate multiple VMs running in the non-secure world.

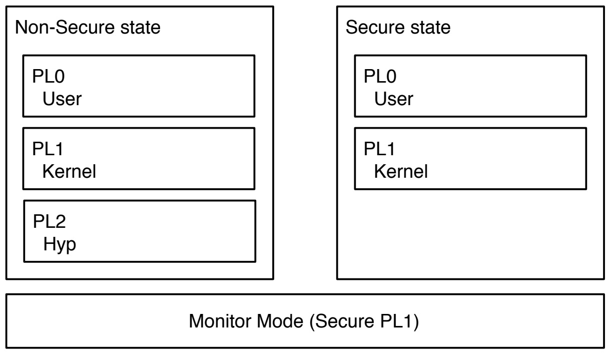


Figure 1: ARMv7 CPU modes.

Hyp mode was introduced as a trap-and-emulate mechanism to support virtualization in the non-secure world. Hyp mode is a CPU mode that is strictly more privileged than other CPU modes, user and kernel modes. Without Hyp mode, the OS kernel running in kernel mode directly manages the hardware and can natively execute sensitive instructions. With Hyp mode enabled, the kernel continues running in kernel mode but the hardware will instead trap into Hyp mode on various sensitive instructions and hardware interrupts. To run VMs, the hypervisor must at least partially reside in Hyp mode. The VM will execute 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 CPU can be switched back into user or kernel mode and the VM can continue executing.

To improve performance, ARM allows many traps to be configured so they trap directly into a VM's kernel mode instead of going through Hyp mode. For example, traps caused by normal system calls or undefined exceptions from user mode can be configured to trap to a VM's kernel mode 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 undefined exception, reducing virtualization overhead.

2.2 Memory Virtualization

In addition to virtualizing the CPU, ARM provides hardware support to virtualize physical memory. When running a VM, the physical addresses managed by the VM are actually *guest physical addresses* and need to be translated into machine addresses, or *host physical addresses*. Similarly to nested page tables on x86, ARM provides a second set of page tables, Stage-2 page tables, configurable from Hyp mode. Stage-2 translation can be completely disabled and enabled from Hyp mode. Stage-2 page tables use ARM's new LPAE page table format, with subtle different requirements than the page tables used by kernel mode.

Similar to standard page tables used to translate virtual addresses to physical addresses used by the VM, ARM virtualization extensions provide a second set of page tables to perform a translation from guest physical addresses to host physical addresses. These Stage-2 translation tables have a format closely resembling, but not identical to, the normal virtual-to-physical page tables, and are stored in normal memory and walked by hardware to translate guest physical addresses into host physical addresses.

Figure 2 shows the complete address translation scheme. Three levels of page tables are used for Stage-1 translation from virtual to guest physical addresses, and four levels of page tables are used for Stage-2 translation from guest to host physical addresses. Stage-2 translation can be entirely enabled or disabled using a bit in the *Hyp Configuration Register* (HCR). The base register for the Stage-2 first-level (L1) page table is specified by the *VirtualizationTranslation Table Base Register* (VTTBR). Both registers are only configurable from Hyp mode.

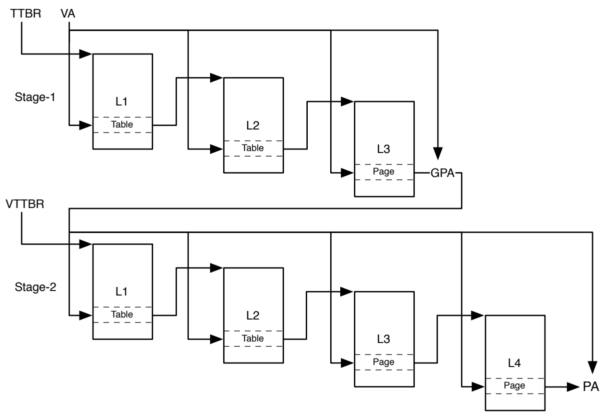


Figure 2: Stage-1 and Stage-2 page table walk on ARMv7 using the LPAE memory long format descriptors. The virtual address (VA) is is first translated into a guest physical address (GPA) and finally into a host physical address (PA).

2.3 Interrupt Virtualization

ARM defines the Generic Interrupt Controller (GIC) architecture [3], which now includes virtualization support (VGIC). It receives interrupts from devices and determines if and on which CPU cores to raise corresponding interrupt signals. The GIC has a distributor and per CPU interfaces. The distributor determines which CPUs receive an interrupt and routes it to the respective CPU interfaces. CPUs access the distributor over a Memory-Mapped I/O (MMIO) interface to enable and set priorities of interrupts. CPU interfaces raise interrupt signals on the respective CPUs and the CPUs access the MMIO CPU interface to acknowledge (ACK) and complete interrupts, known as signaling an End-Of-Interrupt (EOI). Figure 3 shows an overview of the GIC architecture.

Each interrupt signal raised via the CPU interface can be 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 interrupt processing, such as ACKing and EOIing, must now go through the hypervisor.

ARM introduces hardware support for virtual interrupts to reduce the number of traps to Hyp mode. Hardware interrupts trap to Hyp mode to retain hypervisor control, but virtual interrupts trap to kernel mode so that guest OSes can ACK, mask, and EOI them without trapping into the hypervisor. Each CPU has a virtual CPU

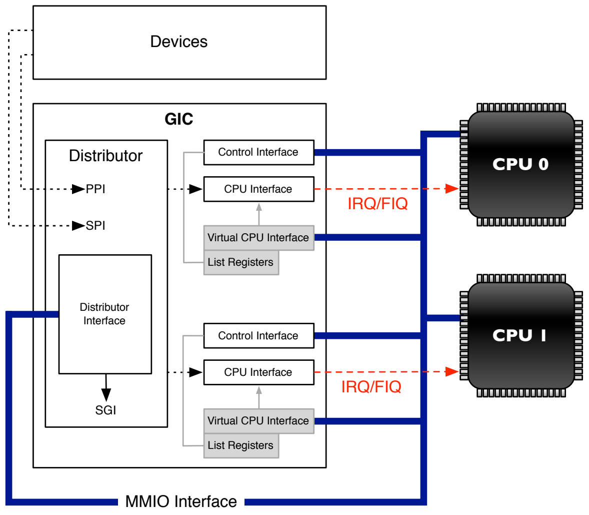


Figure 3: ARM Generic Interrupt Controller (GIC) overview. Devices inject interrupts to the GIC distributor. Interrupts can either be Private Peripheral Interrupts (PPIs), Shared Peripheral Interrupts (SPIs), or Software Generated Interrupts (SGIs). The distributor determines which CPU(s) receives the interrupts and can raise either the IRQ or FIQ line to those CPU(s). CPUs can configure both the CPU interfaces and the distributor using a set of memory mapped configuration registers. The virtualization support consists of a virtual CPU interface, which can be accessed directly from guests and programmed by the hypervisor, to generate virtual interrupts.

interface that the guest OS can interact with through MMIO without trapping to Hyp mode, and a virtual CPU control interface, which can be programmed to raise virtual interrupts using a set of *list registers*, which are only accessed by the hypervisor. In the common case, emulated virtual devices are used to emulate physical devices and when they raise interrupts, these are treated as virtual interrupts, which are programmed into the list registers, causing the GIC to trap the VM directly to kernel mode and lets the guest ACK and EOI the interrupt. Additionally, if a physical device is directly assigned to a VM, the list registers can be programmed so that virtual interrupts generated as a consequence of a physical interrupt from that specific device can be ACKed and EOIed directly by the guest OS instead of needing to trap to the hypervisor.

Virtual interrupts allow guest OSes to handle them once they have been delivered to the CPU, but provide no mechanism to access the distributor. All accesses from VMs to the distributor will cause traps to Hyp mode, and the hypervisor must emulate a virtual distributor. For example, when a guest multicore OS generates a software interrupt that results in an Inter-Processor Interrupt (IPI) between cores, it writes to the *Software Generated Interrupt* (SGI) register on the distributor, which then forwards the interrupt to the corresponding CPU interface. The write to a distributor register will trap to Hyp mode, so the hypervisor can emulate it in software.

2.4 Timer Virtualization

ARM defines the ARM Generic Timers which include support for timer virtualization. Generic timers provide a counter that measures passing of time in real-time, and a timer for each CPU, which is programmed to raise an interrupt to the CPU after a certain amount of time has passed, as per the counter. Timers are likely to be used by both hypervisors and guest OSes, but to provide isolation and retain control, the timers used by the hypervisor cannot be directly configured and manipulated by guest OSes. Such timer accesses from a guest OS would need to trap to Hyp mode, incurring additional overhead for a relatively frequent operation for some workloads. Hypervisors may also wish to virtualize VM time, which is problematic if VMs have direct access to counter hardware.

ARM provides virtualization support for the timers by introducing a new counter, the *virtual counter* and a new timer, the *virtual timer*. A hypervisor can be configured to use physical timers while VMs are configured to use virtual timers. VMs can then access, program, and cancel virtual timers without causing traps to Hyp mode. Access to the physical timer and counter from kernel mode is controlled from Hyp mode, but software running in kernel mode always has access to the virtual timers and counters. Additionally, Hyp mode configures an offset register, which is subtracted from the physical counter and returned as the value when reading the virtual counter.

2.5 Comparison with x86

There are a number of similarities and differences between the ARM virtualization extensions and those for x86 from Intel and AMD. Intel and AMD extensions are very similar, so we just compare ARM and Intel. Both ARM and Intel introduce a new CPU mode for supporting trap and emulate, but they are quite different. ARM's Hyp mode is a separate and strictly more privileged CPU mode than previous user and kernel modes. In contrast, Intel has root and non-root modes [15] which are orthogonal to its CPU protection modes, and can trap operations from non-root to root mode. 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. A hypervisor using ARM's Hyp mode has an arguably simpler set of features to use than the more complex options available with Intel's root mode.

Both ARM and Intel trap into their respective Hyp and root modes, but Intel provides specific hardware support for a VM control block which is automatically saved and restored when switching to and from root mode using only a single instruction. This is used to automatically save and restore guest state when switching between guest and hypervisor execution. In contrast, ARM provides no such hardware support and any state that needs to be saved and restored must be done explicitly in software. This provides some flexibility in what is saved and restored in switching to and from Hyp mode. For example, trapping to ARM's Hyp mode is potentially faster than trapping to Intel's root mode if there is no additional state to save.

ARM and Intel are quite similar in their support for virtualizing physical memory. Both introduce an additional set of page tables is introduced 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 interrupts and timers have no real x86 counterpart. EOIing and masking interrupts by a guest OS on x86 require traps to root mode, whereas ARM's virtual interrupts avoid the cost of trapping to Hyp mode for those interrupt handling mechanisms. Executing similar timer functionality by a guest OS on x86 will incur additional traps to root mode compared to the number of traps to Hyp mode required for ARM. Reading a counter, however, is not a privileged operation on x86 and does not trap, even without virtualization support in the counter hardware.

3 Hypervisor Architecture

KVM/ARM is the first complete open source hypervisor for the ARM architecture. Its relative simplicity and rapid completion was faciliated by specific design choices that allow it to leverage substantial existing infrastructure despite differences in the underlying hardware. Any hypervisor comprises components including core CPU and memory virtualization, a VM scheduler, a memory allocator, an I/O emulation layer, and a management interface. Instead of reinventing and reimplementing such complex core functionality in the hypervisor, and potentially introduce tricky and fatal bugs along the way, KVM/ARM leverages existing infrastructure in the Linux kernel. KVM/ARM builds on the popular KVM interface in the kernel to provide a high degree of code reuse for CPU scheduling, memory management, and device emulation.

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 for many applications. 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. 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 standalone approaches such as Xen [6], which must actively support every platform on which they wish to install the Xen hypervisor.

3.1 Split-mode Virtualization

Simply running a hypervisor entirely in ARM's Hyp mode is attractive since it is the most privileged level. However, since KVM/ARM leverages existing kernel infrastructure such as the scheduler, running KVM/ARM in Hyp mode implies running the 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. The significant changes that would be required to run the kernel in Hyp mode would be very unlikely to be accepted by the Linux kernel community. More importantly, to preserve compatibility with hardware not providing Hyp mode and to run Linux as a guest OS, low-level code would have to be written to work in both modes, 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 the familiar 3GB/1GB split between user address space and kernel address space, Hyp mode uses a single page table register and can therefore not have direct access to the user space portion of the address space. Frequently used functions to access user memory would require the kernel to explicitly map user space data into kernel address space and subsequently perform necessary teardown and TLB maintenance operations, which would result in poor native performance on ARM.

Note that these problems with running a Linux hypervisor using ARM Hyp mode do not occur for x86 hardware virtualization. Because x86 root mode is orthogonal to its CPU privilege modes, it is possible to run the entire Linux kernel in root mode as a hypervisor because the same set of CPU modes available in non-root mode are available in root mode. Nevertheless, given the widespread use of ARM and the advantages of Linux on ARM, finding an efficient virtualization solution for ARM that can leverage Linux and take advantage of the hardware virtualization support is of crucial importance.

KVM/ARM introduces split-mode virtualization, a new approach to hypervisor design that splits the core hypervisor so that it runs across different CPU modes to take advantage of the specific benefits and functionality offered by each CPU mode. As applied to ARM, 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 in Figure 4. 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 one execution context to another as needed, for example switching from a guest execution context to the host execution context. We refer to an execution context as a world, and switching from one world to another as a *world switch*, because the entire state of the system is changed. Since the lowvisor configures the hardware, 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 and can therefore directly leverage existing Linux functionality such as the scheduler. Further, the highvisor 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 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.

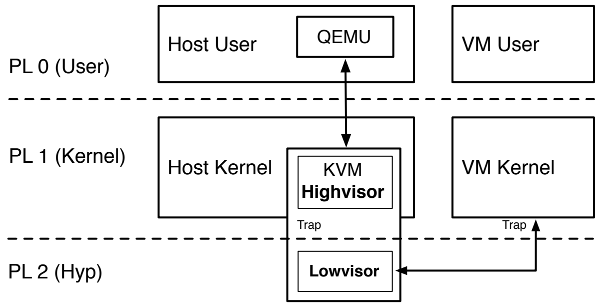


Figure 4: KVM/ARM system architecture.

Because the hypervisor is split across kernel mode and Hyp mode, switching between a VM and the hypervisor involves multiple mode transitions. A trap to the hypervisor while running the VM will first trap to the lowvisor running in Hyp mode. The lowvisor will then cause another trap to run the highvisor. Similarly, going from the hypervisor to a VM when running the highvisor 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 hypervisor. On ARM, the only way to perform these mode transitions to and from Hyp mode is by trapping. However, as shown in Section 5, the cost of this extra trap is not a significant performance cost on ARM.

KVM/ARM uses a memory mapped interface to share data between the highvisor and lowvisor as necessary. Because memory management can be complex, we leverage the highvisor's ability to use the existing memory management subsystem in Linux to manage memory for both the highvisor and lowvisor. Managing the lowvisor's memory involves additional challenges though, because it requires managing Hyp mode's separate address space. 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. Therefore, the highvisor explicitly manages the Hyp mode page tables to map any code executed in Hyp mode and any data structures shared between the highvisor and the lowvisor to the same virtual addresses in Hyp mode and in kernel mode.

Similarly, lowvisor code must also be explicitly mapped to Hyp mode to run in Hyp mode. Extra care must be taken to either link separate C object files, which cannot call standard kernel functions, into sections that can be mapped into Hyp mode, or write all the code running in Hyp mode in assembly to have a single block of code that can be mapped into Hyp mode. Since the lowvisor operations are low-level in nature and would require substantial portions of assembly even if C was used, we opted for the latter approach of writing all lowvisor code in a separate assembly file linked at a specific section by a linker script, making it easy to map it into Hyp mode.

3.2 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, while ensuring that the hypervisor remains in control of the hardware. This involves ensuring that software running in the VM must have *persistent* access to the same register state as software running on the physical CPU, as well as ensuring that physical hardware state associated with the host kernel is persistent across running VMs. Register state not affecting the hypervisor can simply be saved to and restored from memory when switching from a VM to the host and vice versa. KVM/ARM configures access to all other sensitive state to trap, so it can be emulated in the highvisor.

Action	Nr.	State
Save/Restore	38	General Purpose (GP) Registers
	26	Control Registers
	16	VGIC Control Registers
	4	VGIC List Registers
	2	Arch. Timer Control Registers
	32	64-bit VFP registers
	4	32-bit VFP Control Registers
Trap-and-emulate	-	CP14 Trace Registers
	-	WFI Instructions
	-	SMC Instructions
	-	ACTLR Access
	-	Cache ops. by Set/Way
	-	L2CTLR / L2ECTLR Registers

Table 1: *VM and host state on a Cortex-A15 ARMv7 CPU.*

Table 1 shows the state belonging to the hypervisor or a VM, and KVM/ARM's virtualization method for each state category. KVM/ARM performs trap and emulate on sensitive instructions and when accessing hardware state that could affect the hypervisor or would leak information about the hardware to the VM that violates its virtualized abstraction. For example, 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. Because trap and emulate can be expensive, KVM/ARM uses save/restore of registers whenever possible to reduce the frequency of traps by leveraging ARM hardware support for virtualization. For example, access to control registers such as the Stage-1 page table base register is configured not to trap to Hyp mode but are instead saved and restored, allowing a VM to access hardware state directly whenever the hardware supports it and avoiding traps on common guest OS operation such as a context switch.

Register state used by a VM needs to be context switched if it will be used by the hypervisor or another VM. In particular, once a VM traps to the hypervisor, the hypervisor may choose to run another VM. However, in many cases, the hypervisor is likely to run the same VM again. If the register state will not be used by the hypervisor, it can be switched lazily only when another VM is run instead of switching the state on every switch between the hypervisor and the VM. For example, the hypervisor is not likely to perform floating point operations, but there is a significant amount of floating point state in the VFP registers to save and restore. To reduce the cost of trapping to the hypervisor, KVM/ARM does a lazy context switch for floating point state. When switching to a VM from the hypervisor, KVM/ARM configures the hardware to trap on all access to the VFP state. When the VM then accesses VFP state, it traps into the hypervisor, which only then context switches the VFP registers and removes the trap configuration for future accesses by the VM, until the next context switch back to the hypervisor. Since all context switching is done by the lowvisor, no double trap is incurred on a lazy context switch. If the VM does not access VFP registers, KVM/ARM does not need to context switch the VFP registers. Other state may also benefit from lazy context switching, but for simplicity, we have only implemented this functionality for floating point state in the current unoptimized release.

The difference between running inside a VM in kernel or user mode and running the hypervisor in kernel or user mode is determined by how the virtualization extensions have been configured by Hyp mode during the world switch. A world switch from the highvisor to a VM performs the following actions: (1) store all highvisor GP registers on the Hyp stack, (2) configure the VGIC for the VM, (3) configure the timers for the VM, (4) save all highvisor-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 highvisor 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) trap 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. Such an event can be caused by any of the traps mentioned 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 perform another world switch back into the highvisor. The world switch back to the highvisor performs the following actions: (1) store all guest GP registers, (2) disable Stage-2 translation, (3) configure Hyp mode to not trap any register access or instructions, (4) save all VM-specific configuration registers, (5) load the highvisor's configuration registers onto the hardware, (6) configure the timers for the highvisor, (7) save VM-specific VGIC state, (8) restore all highvisor GP registers, (9) trap into kernel mode.

3.3 Memory Virtualization

KVM/ARM provides memory virtualization by ensuring that a VM cannot access physical memory belonging to the hypervisor or other VMs, including any sensitive data. KVM/ARM uses Stage-2 translation to control physical memory access within a VM by configuring the Stage-2 translation page tables to only allow access to certain regions of memory; other accesses will cause Stage-2 page faults which trap to the hypervisor. Since Stage-2 translation can only be configured in Hyp mode, its use is completely transparent to the VM. When the hypervisor performs a world switch to a VM, it enables Stage-2 translation and configures the Stage-2 page table base register accordingly. Since Stage-2 translation is disabled by the lowvisor when switching back to the highvisor, the hypervisor has unfettered access to all physical memory, including the Stage-2 page tables for each VM. Although both the highvisor and VMs can run in kernel mode, 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. Where KVM/ARM handles most Stage-2 page faults by simply calling `get_user_pages` and map the returned page to a guest, a bare-metal hypervisor would be forced to either statically allocate memory to VMs or write a new memory allocation subsystem.

3.4 I/O Virtualization

KVM/ARM leverages existing QEMU and Virtio [22] user space device emulation to provide I/O virtualization. At a hardware level, all I/O mechanisms on the ARM architecture are based on load/store operations to MMIO device regions. With the exception of devices directly assigned to VMs, all hardware MMIO regions are inaccessible from VMs. KVM/ARM uses Stage-2 translations to ensure that physical devices cannot be accessed directly from VMs. Any access outside of RAM regions allocated for the VM will trap to the hypervisor, which can route the access to a specific emulated device in QEMU based on the fault address. 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. As we show in Section 5, KVM/ARM achieves low I/O performance overhead with very little implementation effort.

3.5 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 Hyp mode to trap hardware interrupts to Hyp mode, and performs a world switch to the highvisor to handle the interrupt, so that the hypervisor remains in complete control of hardware resources. When already running in the highvisor, KVM/ARM configures Hyp mode to trap interrupts directly to kernel mode, avoiding the overhead from going through Hyp mode. In both cases, essentially all of the work is done in the highvisor by reusing Linux's existing interrupt handling functionality.

However, VMs must receive notifications in the form of 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 VGIC to inject virtual interrupts into VMs and to reduce the number of traps to Hyp mode. As described in Section 2, virtual interrupts are raised to virtual CPUs by programming the list registers in the MMIO virtual CPU control interface. KVM/ARM configures the Stage-2 page tables to prevent VMs from accessing the control interface, and thereby ensures that only the hypervisor can program the control interface. However, virtual interrupts cannot necessarily be programmed directly into list registers, because another VM may be using the physical CPU, or because there are more virtual interrupts than available list registers. Further, most devices are not aware of the affinity of interrupts to CPU cores, but simply raise interrupts to the GIC distributor. Consequently, a software layer must exist between the emulated devices and the virtual CPU interface.

KVM/ARM introduces the virtual distributor, a software model of the GIC distributor as part of the highvisor. The vitual distributor exposes an interface to user space, so emulated devices in user space can raise virtual interrupts to the virtual 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 the hardware only exposes four list registers, but there are eight pending virtual interrupts, the virtual distributor will choose the four interrupts with highest priority and program the list registers with those four interrupts. Additionally, guest OS software running inside the VM may program the distributor to configure priorities, disable certain interrupts, or to send IPIs to other virtual CPUs. Such operations will trap to the hypervisor and be routed to the virtual distributor, which will populate the effects to its software model or directly generate virtual interrupts through the list registers.

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 necessary when scheduling a different VM to run on a physical core, but unnecessary when simply switching between a VM and the hypervisor since the hypervisor disables all access to the virtual CPU interface when it runs. However, since the virtual distributor is a complicated piece of software with shared state across multiple processes and physical CPUs, the initial unoptimized version of KVM/ARM uses a simplified approach which completely context switches all VGIC state including the list registers on each world switch.

3.6 Timer Virtualization

Reading counters and programming timers are frequent operations in many OSes to manage 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. Since access to the physical timers is controlled using Hyp mode, any software controlling Hyp mode has access to the physical timers. KVM/ARM maintains hardware control by using the physical timers in the hypervisor and disallowing access to physical timers from the VM. We note that unmodified legacy Linux kernels only access the virtual timer and can therefore directly access timer hardware without trapping to the hypervisor.

Unfortunately, due to architectural limitations, the virtual timers cannot directly raise virtual interrupts, but will instead always raise hardware interrupts, which trap to the hypervisor. KVM/ARM detects when a virtual timer programmed by a VM expires, and injects a corresponding virtual interrupt to the VM, performing all hardware ACK and EOI operations in the highvisor. Further, 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 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.

4 Implementation and Adoption

We have successfully integrated our work into the Linux kernel and KVM/ARM is now the standard ARM hypervisor on Linux platforms, as it is included in every kernel beginning with version 3.9. We share some lessons we learned from our experiences in hopes that they may be helpful to others in getting research ideas widely adopted by an existing open source community.

Code maintainability is key. It is a common misconception that a research software implementation providing potential improvements or interesting new features can simply be open sourced and thereby quickly integrated by the open source community. An important point that is often not taken into account 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 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 we prioritized addressing code review comments to make our code suitable for integration into existing systems. An unexpected but important benefit of this decision was that we could leverage the 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. On the one hand one must be well-known to submit code; yet one cannot become known without submitting code. One way to do this is to start small. As part of our work, we also made various small changes to KVM to prepare the support for ARM, which included cleaning up existing code to be more generic and improve cross platform support. The KVM maintainers were glad to accept these small improvements, which generated goodwill and helped us 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, we traveled to ARM headquarters in Cambridge, UK to establish contact with both ARM management and the ARM kernel engineering team, who both contributed to our 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 community puts great efforts into designing extensible and reusable interfaces. Deciding on the appropriateness of an interface is a judgment call and not an exact science. We were fortunate enough to receive help from well-known and respected kernel developers such as Rusty Russell, who helped us drive both the implementation and communication about our interfaces, specifically for the purpose of user space save and restore of registers, a feature useful for both debugging and VM migration. Working with an established developer like Rusty was a tremendous help, both because we could leverage his experience, but also because he has a 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 bootloader must have already 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 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 then simply tests when it starts up whether it is 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 official 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 chain of command. 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's tree with the appropriate approval of the respective architecture maintainers. KVM/ARM, however, 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. Russell King is the ARM kernel maintainer, and Linus pulls directly from his ARM kernel tree for ARM-related code. The situation was particularly interesting, because Russell King did not want to merge virtualization support in the mainline kernel [17] 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 kernel maintainer, 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 in SoC support code was becoming an increasingly big problem for maintainers, and great efforts were underway to reduce board specific code and support a single ARM kernel binary bootable across multiple SoCs. In light of these ongoing changes, getting enough time from ARM kernel maintainers to review the code was challenging, and there was even extra pressure on the maintainers to be highly critical of any new code merged into the ARM tree. Therefore, we had no choice but to keep maintaining and improving the code, and regularly send out updated patch series that followed upstream kernel changes. Eventually, Will Deacon, one of the ARM kernel engineers listed in the ARM kernel maintainers file, made time for several comprehensive and helpful reviews, and after addressing his concerns, he gave us his approval of the code. After all this, when we thought we were done, we finally received some feedback from Russell King.

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 used by older Linux kernels do not populate such a register. KVM/ARM therefore loads the instruction from memory and decodes 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 had no choice but to abandon the otherwise well-liked and useful code base. We can only speculate about the true motives behind this decision, as the ARM kernel maintainers would not engage in a discussion about the subject.

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. In getting all these things to come together in the end before the 3.9 merge window, the key was having a good relationship with many of the kernel developers to get their help, and being persistent in continuing to push to have the code merged in the face of various challenges.

5 Experimental Results

We present some experimental results that quantify the performance of KVM/ARM on real multicore ARM hardware. We evaluate the virtualization overhead of KVM/ARM compared to direct execution by running both microbenchmarks and real application workloads within VMs and directly on the hardware. We measure both the performance and energy virtualization costs of using KVM/ARM. We also compare the virtualization and implementation costs of KVM/ARM versus KVM x86 to demonstrate the effectiveness of KVM/ARM against a more mature hardware virtualization platform. These results provide the first real hardware measurements of the performance of ARM hardware virtualization support as well as the first comparison between ARM and x86.

5.1 Methodology

We used one ARM and two x86 platforms for our measurements. ARM measurements were obtained using an Insignal Arndale board [14] 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 1256v2 CPU, two physical SSD drives of which only one was used, and 1GB Ethernet connected to a 100Mb network infrastructure.

Given the differences in hardware platforms, our focus was not on measuring absolute performance, but rather the relative performance differences between virtualized and direct execution on each platform. Since our goal is to evaluate hypervisors, 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 the same as much as possible. Both the host and guest VMs on all platforms were Ubuntu version 12.10. Because the Linux 3.9 kernel is not yet released as of the time of this writing, device support is not complete and therefore it does not run across all of the hardware platforms we used. As a result, we used the Linux 3.6 kernel for our experiments, with patches for KVM/ARM rebased 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. 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 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.

apache	Apache v2.2.22 Web server running ApacheBench v2.3, 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` [19] using unix domain sockets and 100 process groups running with 500 loops

Table 2: Benchmark applications workloads.

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 someone else, 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 using custom small guest OSes only written for testing aspects of the respective systems. 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 [18] v3.0 on both single and multicore. 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 running both natively and in VMs, and compared the normalized performance across ARM and x86 hypervisors. Table 2 shows the applications 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 [5] and measure 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 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.

5.2 Performance Measurements

Micro Test	ARM	ARM no vgic/vtimers	x86 laptop	x86 server
Hypercall	4,917	2,112	1,263	1,642
Trap	27	27	632	821
I/O Kernel	6,248	2,945	2,575	3,049
I/O User	6,908	3,971	8,226	10,356
IPI	10,534	-	13,670	16,649
EOI	9	-	1,713	2,195

Table 3: Micro-architectural cycle counts.

Table 3 presents various micro-architectural costs of virtualization using KVM/ARM on ARM and KVM x86 on 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; available x86 hardware does not yet provide such mechanism. The ARM no VGIC/vtimers shows measurements without the cost of saving and restoring VGIC state and 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 world switch costs are only about 500 cycles more than the hardware accelerated world switch cost on the x86 server platform.

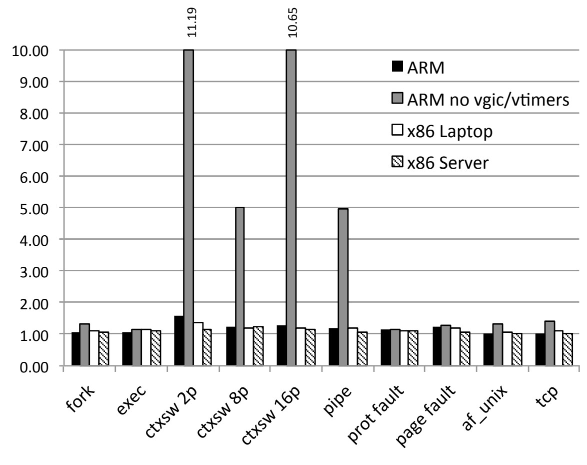
Trap is the cost of switching the hardware mode from the VM into the respective CPU mode for running the hypervisor, Hyp mode on ARM and root mode on x86. ARM is much faster than x86 because it only needs to manipulate two registers to perform this trap, whereas the cost of a trap on x86 is the same 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 (`inl` on x86, MMIO load on ARM) to a device, which is emulated inside the kernel and returning to the VM. I/O User shows the cost of issuing an I/O read operation from a device emulated in user space, adding to I/O Kernel the cost of transitioning from the kernel to a user space process on the host for I/O. This is representative of the cost of using QEMU. Since these operations involve world switches, saving and restoring VGIC state is again a significant cost on ARM. KVM x86 is much faster than KVM/ARM on I/O Kernel, but 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 than on ARM.

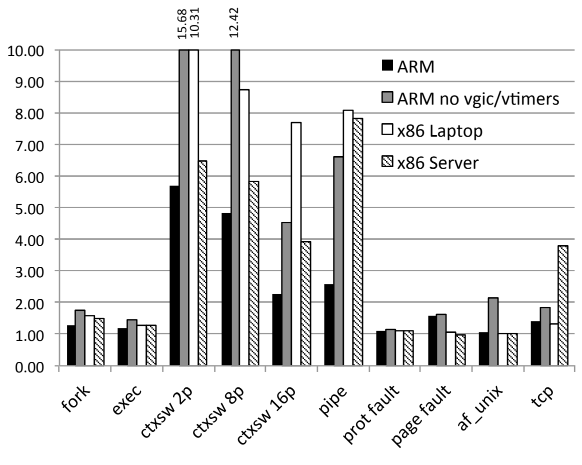
IPI is the cost of issuing an IPI to another virtual CPU core when both virtual cores are running on separate physical cores and both are actively running inside the VM. The IPI test measures the time starting from sending an IPI until the other virtual core responds to the IPI; it does not include time spent by the other core signaling completion of the interrupt to the interrupt controller. ARM is somewhat faster because the underlying hardware IPI on x86 is expensive, and x86 APIC MMIO operations require KVM x86 to perform instruction decoding not needed on ARM.

EOI is the cost of completing an interrupt on both platforms. It includes both interrupt acknowledgment and completion 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 200 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.

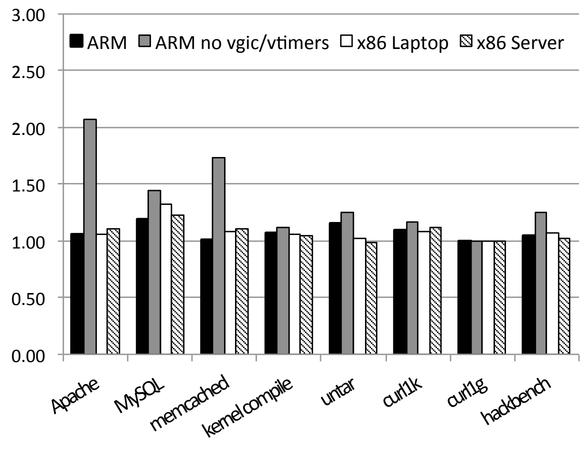
Figures 5a and 5b show normalized performance for running lmbench in a VM versus running directly on the host. Figure 5a shows that KVM/ARM and KVM x86 have similar virtualization overhead in a single core configuration. For comparison, we also show KVM/ARM performance without VGIC/vtimers. Overall, using VGIC/vtimers provides slightly better performance except for the pipe and context switch workloads where the difference between using and not using VGIC/vtimers is substantial. The high overhead without VGIC/vtimers in these cases 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



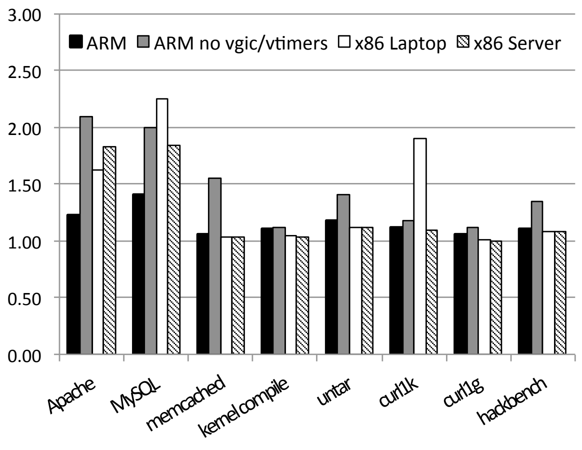
(a) UP VM normalized lmbench performance



(b) SMP VM normalized lmbench performance



(c) UP VM normalized application performance



(d) SMP VM normalized application performance

Figure 5: UP and SMP performance measurements. All results are normalized virtual relative to native with lower being less overhead.

workload with VGIC support, but without vtimers, and we counted the number of timer read exits when running without vtimers support.

Figure 5b 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 page faults. Both systems have the worst overhead for the pipe and context switch workloads, though KVM x86 is more than three times worse for pipe. These differences are due to the cost of repeatedly sending an IPI from the sender of the data in the pipe to the receiver for each message. x86 not only has higher IPI overhead than ARM, but it must also EOI 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 EOI the IPIs.

Figures 5c and 5d show normalized performance for running application workloads in a VM versus running directly on the host. Figure 5c shows that KVM/ARM and KVM x86 have similar virtualization overhead across all workloads in a single core configuration, but Figure 5d 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 most application workloads, and the highest virtualization overheads for Apache and MySQL are still significantly less than the more mature KVM x86 system. 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 than without ARM VGIC/vtimers support. Using VGIC/vtimers to reduce the number of world switches outweighs the additional world switch overhead from saving and restoring VGIC/vtimer state.

5.3 Power Measurements

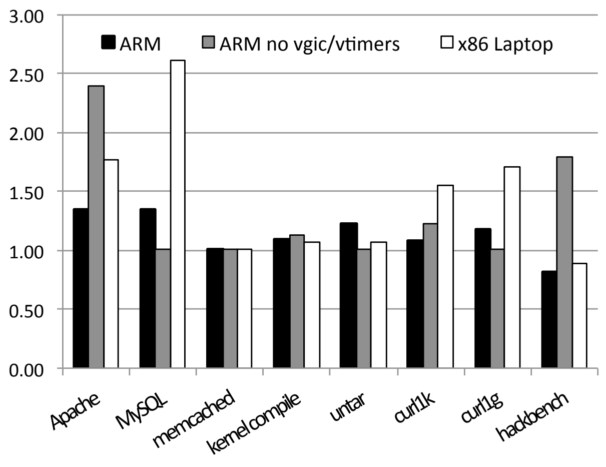


Figure 6: SMP VM normalized energy consumption.

Figure 6 shows normalized power consumption of using virtualization versus direct execution for various application workloads. We only compared KVM/ARM on ARM against KVM x86 on x86 laptop. The Intel Core i7 CPU used here is one of Intel's more power optimized processors, and we expect that server power consumption would be even higher. The measurements show that KVM/ARM using VGIC/vtimers is more power efficient than KVM x86 virtualization in all cases except kernel compile. This is most likely explained by the CPU intensive operation of compilation compared to the otherwise more balanced workloads, which indicates that ARM is more optimized considering an entire SoC, where x86 suffers more extreme power consumption, when for example being memory bound. While a more detailed study of energy aspects of virtualization are beyond the scope of this paper, these measurements nevertheless provide useful data comparing ARM and x86 virtualization energy costs.

5.4 Implementation Complexity

We compare the code complexity of KVM/ARM to its KVM x86 counterpart. KVM/ARM is 5,622 lines of code (LOC), counting just the architecture-specific code added to Linux to implement it, of which the lowvisor is a mere 754 LOC. KVM/ARM's LOC is less than partially complete bare-metal microvisors written for Hyp mode [24], with the lowvisor LOC almost an order of magnitude smaller. As a conservative comparison, KVM x86 is 24,674 LOC, excluding guest performance monitoring support, not yet supported by KVM/ARM, and 3,305 LOC required for AMD support. These numbers do not include KVM's architecture-generic code, 5,978 LOC, which is shared by all systems. Table 4 shows a breakdown of the total architecture-specific code into its major components.

Component	**KVM/ARM**	**KVM x86** (Intel)
Core CPU	1,876	15,691
Page Fault Handling	673	4,261
Interrupts	1,194	2,151
Timers	221	630
Other	1,658	1,941
Architecture-specific	5,622	24,674

Table 4: 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 have 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. KVM/ARM's out-of-tree MMIO instruction decode implementation was much simpler, only 462 LOC. (4) The various paging mode on x86 requires more software logic to handle page faults. (5) x86 requires more software logic to support interrupts and timers than ARM, which provides VGIC/vtimers hardware support that reduces software complexity.

6 Recommendations

From our experiences building KVM/ARM, we offer a few recommendations for hardware designers to simplify and optimize future hypervisor implementations.

Share kernel mode memory model. The hardware mode to run a hypervisor should use the same memory model as the hardware mode to run OS kernels. Software designers then have greater flexibility in deciding how tightly to integrate a hypervisor with existing OS kernels. ARM Hyp mode unfortunately did not do this, preventing KVM/ARM from simply reusing the kernel's page tables in Hyp mode. This reuse would have simplified the implementation and allowed for performance critical emulation code to run in Hyp mode, avoiding a complete world switch in some cases. Some might argue that this recommendation makes for more complicated standalone hypervisor implementations, but this is not really true. For example, ARM kernel mode already has a simple option to use one or two page table base registers to unify or split the address space. Our recommendation is different from the x86 virtualization approach, which does not have a separate and more privileged, hypervisor CPU mode. Having a separate CPU mode has the potential for improved performance for some hypervisor designs, but without sharing the kernel memory model, makes other common hypervisor designs difficult.

Make VGIC state access fast, or at least infrequent. While VGIC support can improve performance especially on multicore systems, our measurements also show that access to VGIC state adds substantial overhead to world switches. This is caused by slow MMIO access to the VGIC control interface in the critical path. Improving the MMIO access time is likely to improve VM performance, but if this is not possible or cost-effective, MMIO accesses to the VGIC could at least be made less frequent. For example, a summary register could be introduced describing the state of each virtual interrupt. This could be read when performing a world switch from the VM to the hypervisor to get information which can currently only be obtained by reading all the list registers (see Section 3.5) on each world switch.

Completely avoid IPI traps. Hardware support to send virtual IPIs directly from VMs without the need to trap to the hypervisor would improve performance. Hardware designers underestimate how frequent IPIs are on modern multicore OSes, and our measurements reveal that sending IPIs adds significant overhead for some workloads. The current VGIC design requires a trap to the hypervisor to emulate access to the IPI register in the distributor, and this emulated access must be synchronized between virtual cores using a software locking mechanism, which adds significant overhead for IPIs. Current hardware supports receiving the virtual IPIs, which can be ACKed and EOIed without traps, but unfortunately does not address the also important issue of sending virtual IPIs.

Make virtual timers transparent and directly support virtual interrupts. While the virtual counter avoids the need to trap to the hypervisor when VMs need to update a counter, software written to read the physical counter will trap to hypervisors that virtualize time. Instead of providing two separate counters with a separate access mechanism, the hardware should expose a single interface to access a counter, and Hyp mode should decide whether such an operation reads the virtual or physical counter.

Since virtual timers are programmed by VMs for use by VMs, it should not be necessary to trap to the hypervisor when a virtual timer expires. By instrumenting KVM/ARM while running the workloads discusses in Section 5, we were able to determine that while programming a timer is not as frequent as reading a counter, it is still a fairly common operation, and supporting this feature would be like to both simplify hypervisor design and improve performance.

Expect hypervisors to swap. When hypervisors swap pages to disk the underlying physical mappings of memory pages change, without the knowledge of guest OSes, which can cause cache conflicts. For example, consider a standard page containing executable guest code. When this page is swapped to disk, its guest physical to host physical mapping is invalidated in the Stage-2 translation tables by the hypervisor so that memory accesses to that page occurring in the future will trap to the hypervisor. When this happens, the hypervisor will allocate another physical memory page and fill that page with the code that was previously swapped out. However, since the page that was just swapped in, may have previously been used by another VM and may have contained code mapped at the same virtual address, an entry with the same signature may already exist in the instruction cache containing invalid code, and the VM may end up executing the wrong instructions.

To address this problem, hardware support should be provided so that hypervisors can perform cache maintenance operations on a page-per-page basis using physical page addresses. For example, one solution would be to simply invalidate the entry for that page in the instruction cache when swapping in the page. However, current hardware does not necessarily allow this operation, depending on the type of hardware cache that is used in the SoC. For example, the ARM architecture allows SoCs to use Virtually-Indexed-Physically-Tagged (VIPT) instruction caches, and in this case, it is not possible to identify all potential mappings of guest virtual addresses to the physical address of a given page. As a result, it becomes necessary to flush the entire cache for correctness in the presence of swapping, resulting in a very undesirable performance cost. To avoid this cost, since hypervisors are not aware of all possible virtual addresses used to access that physical page, it must be possible for hypervisors to perform cache maintenance operations on a page-per-page basis using physical page addresses.

Prepare for memory deduplication. Memory deduplication is a commonly used technique to reduce physical memory pressure in virtualized environments [25]. It works by letting two or more VMs have a mapping to the same physical memory page, and marking all such mappings read-only in the Stage-2 page tables. If a VM tries to write to such a page, a permission fault will trap to the hypervisor, which can perform copy-on-write to copy the page content to a new page, which can be mapped writable to the VM. However, the ARM hardware support for virtualization does not report the GPA on permission faults; it only reports the GPA on a normal page fault, where the page is not present. KVM/ARM does support memory deduplication by leveraging Linux's built-in KSM feature, but the software is unnecessarily complicated, because the GPA has to be manually resolved by walking the guest's Stage-1 page tables when taking permission faults inside the lowvisor. Further, on multicore VMs, another virtual CPU may modify the Stage-1 page tables in the process, and the lowvisor must detect such races and handle them accordingly. Since memory deduplication is likely to be used by any hypervisor, hardware designers should consider the typical path of copy-on-write handling and provision the hardware accordingly. In the case of ARM, this should include populating the GPA fault address register (HPFAR) on permission faults.

7 Related Work

Virtualization has a long history [20], but has enjoyed a resurgence starting in the late 1990s. Most efforts have almost exclusively focused on virtualizing the x86 architecture. While systems such as VMware [1, 8] and Xen [6] were originally based on software-only approaches before the introduction of x86 hardware virtualization support, all x86 virtualization platforms, VMware, Xen, and KVM [16], now leverage x86 hardware virtualization support. Because x86 hardware virtualization support differs substantially from ARM in the ability to completely run the hypervisor in the same mode as the kernel, x86 virtualization approaches do not lend themselves directly to take advantage of ARM hardware virtualization support.

Some x86 approaches also leverage the host kernel to provide functionality for the hypervisor. VMware Workstation's hypervisor makes use of host kernel mechanisms and device drivers, but cannot reuse host kernel code since it is not integrated with the kernel, resulting in a more complex hypervisor to build and maintain. 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.

Full-system virtualization of the ARM architecture is a relatively unexplored research area. Most approaches are software only. A number of bare metal hypervisors have been developed [11, 12, 21], but these are not widespread, are developed specifically for the embedded market, and must be modified and ported to every single host hardware platform, limiting their adoption. An abandoned port of Xen for ARM [13] requires comprehensive modifications to the guest kernel, and was never fully developed. VMware Horizon Mobile [7] uses hosted virtualization to leverage Linux's support for a wide range of hardware platforms, but requires modifications to guest OSes and its performance is unproven. An earlier prototype for KVM on ARM [9, 10] used an automated lightweight paravirtualization approach to automatically patch kernel source code to run as a guest kernel, but was developed prior to the introduction of hardware support for virtualization on ARM. None of these paravirtualization approaches could run unmodified guest OSes.

With the introduction of ARM hardware virtualization support, Varanasi and Heiser performed a study to roughly estimate its performance using a software simulator and an early prototype of a custom hypervisor lacking important features like SMP support and use of storage and network devices by multiple VMs [24]. Because of the lack of hardware or a cycle-accurate simulator, no real performance evaluation was possible. In contrast, we present the first evaluation of ARM virtualization extensions using real hardware, provide a direct comparison with x86, and present the design and implementation of a complete hypervisor using ARM virtualization extensions, including SMP support. Our conclusions based on a complete hypervisor running on real hardware differ from this previous study in terms of the simplicity of designing a hypervisor using Hyp mode and the real cost of hypervisor operations. We further show how SMP support introduces performance challenges and the benefits of hardware support for virtualized timers, issues not addressed by the previous study.

In parallel with our efforts, a newer version of Xen exclusively targeting servers [26] is being developed using ARM hardware virtualization support. Because Xen is a bare metal hypervisor that does not leverage kernel functionality, it can be architected to run entirely in Hyp mode rather than using split-mode virtualization. At the same time, this requires a substantial commercial engineering effort and Xen ARM still does not have SMP support. We would have liked to have done a direct performance comparison between Xen and KVM/ARM, but Xen still does not work on the popular Arndale board used for our experiments. 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 XenBus paravirtualized drivers. In contrast, KVM/ARM uses standard Linux components to enable faster development, full SMP support, and the ability to run unmodified OSes. KVM/ARM is easily supported on new devices with Linux support, and we spent almost no effort to support KVM/ARM on ARM's Versatile Express boards, the Arndale board, and hardware emulators. While Xen can potentially reduce world switch times for operations that can be handled inside the Xen hypervisor, switching to Dom0 for I/O support or switching to other VMs would involve a context switching of the same state as KVM/ARM.

Microkernel approaches for hypervisors [23, 11] 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. Split-mode virtualization also provides a different split of hypervisor functionality. KVM/ARM's lowvisor is a much smaller code base that implements only the lowest level hypervisor mechanisms. It does not include higher-level functionality present in the hypervisor TCB used in these other approaches.

8 Conclusions

KVM/ARM is the first full system ARM virtualization solution that can run unmodified guest operating systems on ARM multicore hardware. By introducing split-mode virtualization, it can leverage ARM virtualization support while leveraging Linux kernel mechanisms and code to simplify hypervisor development and maintainability. Our experimental results show that KVM/ARM (1) incurs minimal performance impact from the extra traps incurred by split-mode virtualization, (2) has modest virtualization overhead and power costs, within 10% of direct native execution on multicore hardware for most application workloads, and (3) can provide significantly lower virtualization overhead and power costs compared to the widely-used KVM x86 virtualization on multicore hardware. We have integrated KVM/ARM into the mainline Linux kernel, ensuring its wide adoption as the virtualization platform of choice for ARM.

9 Acknowledgments

KVM/ARM benefited enormously from the help of the Linux community. Marc Zyngier implemented VGIC and virtual timers support, provided invaluable insight in understanding their impact on virtualization performance, and helped with getting KVM/ARM running on early ARM Cortex-A15 hardware. Peter Maydell provided QEMU support and useful insights in understanding the ARM architecture. Rusty Russell implemented parts of the world switch and instruction emulation mechanisms. Will Deacon provided helpful code reviews. Retired and current KVM maintainers, Avi Kivity, Gleb Natapov, and Marcelo Tosatti, provided helpful code reviews and insight in understanding the KVM x86 implementation.

Keith Adams, Hani Jamjoom, Oren Laadan, Erran Li, and Emmett Witchel provided helpful comments on earlier drafts of this paper. Richard Phelan made it possible for us to experiment with early ARM Cortex-A15 hardware. This work was supported in part by ARM, a Google Research Award, and NSF grants CNS-1162447, CNS-1018355, and CNS-0905246.

References

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

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

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

[4] ARM Ltd. ARM Architecture Reference Manual ARMv7-A DDI0406C.b, 2012.

[5] ARM Ltd. ARM Energy Probe, Mar. 2013. `http://www.arm.com/products/tools/arm-energy-probe.php`.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, Dec. 2003.

[7] K. Barr, P. Bungale, S. Deasy, V. Gyuris, P. Hung, C. Newell, H. Tuch, and B. Zoppis. The VMware Mobile Virtualization Platform: is that a hypervisor in your pocket? *SIGOPS Operating Systems Review*, 44:124–135, Dec. 2010.

[8] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang. Bringing Virtualization to the

x86 Architecture with the Original VMware Workstation. *ACM Transactions on Computer Systems*, 30:12:1–12:51, Nov. 2012.

[9] C. Dall and J. Nieh. KVM for ARM. In *Proceedings of the Ottawa Linux Symposium*, July 2010.

[10] J.-H. Ding, C.-J. Lin, P.-H. Chang, C.-H. Tsang, W.-C. Hsu, and Y.-C. Chung. ARMvisor: System Virtualization for ARM. In *Proceedings of the Ottawa Linux Symposium*, June 2012.

[11] General Dynamics. OKL4 Microvisor, Mar. 2013. `http://www.ok-labs.com/products/okl4-microvisor`.

[12] Green Hills Software. INTEGRITY Secure Virtualization, Mar. 2013. `http://www.ghs.com`.

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

[14] InSignal Co.,Ltd. ArndaleBoard.org, Mar. 2013. `http://arndaleboard.org`.

[15] Intel Corporation. Intel 64 and IA-32 Architectures Software Developers Manual, 325462-044US, 2011.

[16] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. **`kvm`**: The Linux Virtual Machine Monitor. In *Proceedings of the Ottawa Linux Symposium*, June 2007.

[17] Linux ARM Kernel Mailing List. A15 h/w virtualization support, Apr. 2011. `http://archive.arm.linux.org.uk/lurker/message/20110412.204714.a36702d9.en.html`.

[18] L. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *Proceedings of the 1996 USENIX Annual Technical Conference*, pages 23–23, Jan. 1996.

[19] I. Molnar. Hackbench, Feb. 2013. `http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c`.

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

[21] Red Bend Software. vLogix Mobile, Mar. 2013. `http://www.redbend.com/en/mobile-virtualization`.

[22] R. Russell. virtio: Towards a De-Facto Standard For Virtual I/O Devices. *SIGOPS Operating Systems Review*, 42:95–103, July 2008.

[23] U. Steinberg and B. Kauer. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proceedings of the 5th European Conference on Computer Systems*, pages 209–222, Apr. 2010.

[24] P. Varanasi and G. Heiser. Hardware-Supported Virtualization on ARM. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems*, pages 11:1–11:5, July 2011.

[25] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. *SIGOPS Operating Systems Review*, 36:181–194, Dec. 2002.

[26] Xen.org. Xen ARM, Mar. 2013. `http://xen.org/products/xen_arm.html`.