KVM for ARM

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

Abstract

As ARM CPUs grow in performance and ubiquity across phones, netbooks, and embedded computers, providing virtualization support for ARM-based devices is increasingly important. We present KVM/ARM, a KVM-based virtualization solution for ARM-based devices that can run virtual machines with nearly unmodified operating systems. Because ARM is not virtualizable, KVM/ARM uses lightweight paravirtualization, a script-based method to automatically modify the source code of an operating system kernel to allow it to run in a virtual machine. Lightweight paravirtualization is architecture specific, but operating system independent. It is minimally intrusive, completely automated, and requires no knowledge or understanding of the guest operating system kernel code. By leveraging KVM, which is an intrinsic part of the Linux kernel, KVM/ARM's code base can be always kept in line with new kernel releases without additional maintenance costs, and can be easily included in most Linux distributions. We have implemented a KVM/ARM prototype based on the Linux kernel used in Google Android, and demonstrated its ability to successfully run nearly unmodified Linux guest operating systems.

1 Introduction

To provide the benefits of virtualization to Linux users, Kernel Virtual Machine (KVM) has been included in Linux starting with kernel version 2.6.20. Its tremendous success is in large part due to its open-source distribution and its relative simplicity compared to other approaches. This simplicity is achieved by leveraging the functionality already provided by the Linux kernel, and relying on some level of hardware virtualization support. KVM runs on a wide range of architectures, currently providing full support for x86 and PowerPC, and experimental support for Itanium (ia64) and s390. The x86 and ia64 implementations rely on hardware virtualization extensions and the PowerPC and s390 architectures are virtualizable.

Unfortunately, KVM does not support the ARM architecture, which is increasingly ubiquitous. While ARM is known for excellent power consumption, small die size, and compact code, recent CPUs based on the ARM architecture are also quite powerful, and are being incorporated in growing numbers into a wide range of products. Mobile phones are almost exclusively based on ARM, and ARM-based tablets and laptops with 3G connections are increasing in popularity. Users increasingly expect these devices to be able to perform a multitude of tasks, including browse the Internet, play games, and run thousands of other applications from an online application store. ARM Linux is becoming more important with the introduction of several Linux-based distributions targeting mobile and embedded ARM-based devices, Google Android being one of them.

The key challenge in providing virtualization on ARM is that the ARM architecture is not virtualizable. A virtualizable architecture would allow a virtual machine to directly execute on the real hardware while guaranteeing that the virtual machine monitor (VMM) retains control of the CPU. This is done by running the operating system in the virtual machine, the guest operating system, in non-privileged mode while the VMM runs in privileged mode. ARM is not virtualizable because there are a number of sensitive instructions, used by operating systems, which do not generate a trap when executed in non-privileged mode. Their behavior is either unpredictable or they behave differently, causing an operating system that uses these sensitive instructions to not run correctly if run in a virtual machine directly executed on the real hardware. There is also no hardware virtualization support on ARM. The result is that ARM CPU and memory virtualization are difficult.

We present KVM for ARM (KVM/ARM), a KVM-based virtualization solution for ARM that runs nearly unmodified operating system instances in virtual machines. KVM/ARM retains the simplicity of the KVM architecture in the absence of ARM hardware virtualization support by introducing *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 KVM instead of issuing sensitive instructions to enable a *trap-and-emulate* virtualization solution. Lightweight paravirtualization is architecture specific, but operating system independent. It is completely automated and requires no knowledge or understanding of the guest operating system kernel code. This is in stark contrast to 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 virtual machines.

This paper presents the design and implementation of KVM/ARM. Section 3 describes KVM/ARM CPU virtualization using lightweight paravirtualization. Section 4 describes KVM/ARM memory virtualization. Section 5 describes the current implementation status of KVM/ARM and ideas for improvement. Finally, we present some concluding remarks.

2 Related work

Virtualization has been around since the of the 1970s [6], but re-emerged in the 1990s as commodity x86 hardware became fast enough to run multiple operating systems simultaneously. The x86 architecture was previously not virtualizable since many sensitive instructions did not trap when executed in non-privileged mode [1]. Emulation could be used where each guest instruction is interpreted in software, but this is too slow for practical use. To overcome this problem, VMware [13] introduced efficient dynamic binary translation mechanisms to translate sensitive instructions to other instructions to enable x86 virtualization with low performance overhead. However, the dynamic binary translation mechanisms are not easy to implement, resulting in a complex solution that is likely to be too heavyweight to use for more resource-constrained mobile devices such as smartphones. Xen [5] used paravirtualization [17] to provide x86 virtualization, in which guest operating systems are extensively modified by hand to use a rich set of hypercalls in lieu of sensitive instructions. However, paravirtualization cannot run existing unmodified operating systems, and the modifications are extensive enough that supported guest operating systems lag significantly behind the latest available unmodified operating system versions.

Intel and AMD have recently begun equipping x86 CPUs with native hardware virtualization support, Intel VT [9] and AMD-V [2], respectively. A new CPU guest mode is provided for running virtual machines such that sensitive instructions automatically trap so they can be handled by a VMM, and nested page tables provide hardware translation between physical memory addresses perceived by the guest operating system and host physical addresses on the real hardware through a data structure managed by the VMM. KVM leverages hardware virtualization support for x86 CPUs together with existing Linux kernel functionality to provide a relatively simple virtualization solution compared to VMware and Xen. KVM implements a simple kernel module, which provides full native virtualization supporting completely unmodified guest operating systems. Unfortunately, no such hardware virtualization support exists for ARM. KVM/ARM is designed to preserve the simplicity of KVM as much as possible while enabling virtualization support on a non-virtualizable architecture.

The growing ubiquity of ARM CPUs and continued advances in their performance have spurred various efforts to provide virtualization on ARM. Several commercial solutions are being developed, including VLX for ARM by VirtualLogix [15], OKL4 Microvisor by OK Labs [11], MVP by VMware [16], and INTEGRITY secure virtualization by Green Hills [7]. None of these solutions are open-source and all of them require paravirtualization.

Xen ARM [8] is the only other open-source ARM virtualization approach available. Xen paravirtualization requires access to guest operating system source code and maintenance of changes to each version of the source tree. The price of paravirtualization is increased maintenance cost and more limited availability in terms of supported guest operating system versions. For example, Xen ARM requires modifying by hand approximately 4500 lines of code in the guest operating system [14]. The most recent kernel version it can support in a guest operating system is a modified Linux 2.6.11 kernel, a relatively old version of Linux. In contrast, KVM/ARM's lightweight paravirtualization requires minimal modifications to guest operating systems, and those modifications are simple enough that they can be completely automated by a script. This makes it relatively easy for KVM/ARM to support more recent versions of guest operating systems.

3 CPU virtualization

Virtual machines must not be allowed to access the privileged state of the physical CPU and thereby gain unwanted control of hardware resources. Therefore, guest operating systems must always run in a non-privileged mode. The non-privileged mode on ARM is called *user mode*.

Popek and Goldberg [12] define sensitive instructions as the group of instructions where the effect of their execution depends on the mode of the processor or the location of the instruction in physical memory. A sensitive instruction is also privileged if it always generates a trap, when executed in user mode. The VMM can only guarantee correct guest execution without the use of dynamic translation if all sensitive instructions are also privileged. In other words, an architecture is virtualizable if and only if the set of sensitive instructions is a subset of the set of privileged instructions. If that is the case, the VMM can be implemented using a classic trap-and-emulate solution. Unfortunately, ARM is not virtualizable as the architecture defines both sensitive privileged instructions and sensitive non-privileged instructions.

The sensitive privileged instructions defined by the ARM architecture are the *coprocessor access instructions* which are used to access the *coprocessor interface*. There is no such thing as a physical coprocessor, but the semantics are used merely to extend the instruction set by transferring data between general purpose registers and registers belonging to one of the sixteen possible coprocessors. The architecture always defines coprocessor number 15 which is called the *system control coprocessor* and controls the virtual memory system. Specific implementations of the ARM architecture can define other coprocessors to allow software to access special hardware or otherwise leverage additional hardware logic. For instance, coprocessor 14 is often used to access floating point hardware. The coprocessor access instructions are: `CDP`, `LDC`, `MCR`, `MCRR`, `MRC`, `MRRC`, and `STC`. These instructions do not have to be handled specially as they trap when they are executed in user mode. When that happens, KVM/ARM catches the trap and emulates the sensitive privileged instruction in software.

The ARM architecture also defines sensitive non-privileged instructions which cannot be handled using just trap and emulate because they do not trap. These instructions deal with processor modes, status registers, and memory accesses that depend on CPU mode.

Processor mode instructions relate to ARM's 7 processor modes: user mode and 6 privileged modes.[1] Each mode has a number of *banked registers*, which means that, for instance, register 13 points to a different physical register in *supervisor mode* than in user mode.[2] Specific versions of *load/store multiple instructions* access user mode registers even when the processor is in a privileged mode. When executed in user mode, these instructions do not trap and are therefore sensitive and non-privileged. These instructions are the `LDM(2)` and `STM(2)` instructions.

Status register instructions relate to special ARM status registers. ARM processors have a special register called the *Current Program Status Register* (CPSR), which specifies the current mode of the CPU and other state information. Some of the bits in the CPSR are privileged, such as the mode bits, and some are accessible in user mode. Five of the privileged modes also have a banked *Saved Program Status Register* (SPSR), which contains a copy of the user mode CPSR as it was when the processor entered the privileged mode.[3]. The *status register access instructions* `CPS,MRS,MSR,RFE`, `SRS` read and write the CPSR and the SPSR. Writes to privileged bits are ignored when the CPU is in user mode and access to the SPSR is unpredictable in user mode. Further, almost all data processing instructions exist in a special mode, which replaces the content of the CPSR with that of the SPSR. These instructions are denoted by appending an S to the instruction name: `ADCS`, `ADDS`, `ANDS`, `BICS`, `EORS`, `MOVS`, `MVNS`, `ORRS`, `RSBS`, `RSCS`, `SBCS`, and `SUBS`. Likewise, the `LDM(3)` instruction replaces the content of

[1] See pages A2-3 to A2-5 in the ARM Architecture Reference Manual [3] for more information

[2] The differences between the privileged modes only concern the banked registers and can be ignored throughout this paper.

[3] Actually, the CPSR is only copied to the SPSR when entering privileged mode through exceptions and not when manually switching modes

the CPSR with that of the SPSR in addition to loading multiple registers. The behavior of these instructions is unpredictable when executed in user mode and the instructions are therefore all sensitive and non-privileged.

Memory access instructions that depend on CPU mode relate to access protection. The virtual memory system on ARM processors uses access protection bits to limit access to memory depending on the CPU mode. Regular memory accesses are *not* sensitive according to Goldberg and Popek, as they will trap when executed in a less privileged mode (reduced memory access rights). However, the architecture defines a number of instructions that access memory using user mode access permissions even though the CPU is in a privileged mode. These instructions are called *Load/Store with translation* and there are four of them: `LDRBT`, `LDRT`, `STRBT`, and `STRT`. When executed in user mode, these instructions behave as regular memory access instructions. Thus the effect of executing these instructions depends on the mode of the processor and the instructions do not trap due to memory access violations. They are therefore sensitive and non-privileged.

3.1 Lightweight paravirtualization

To avoid the problems with sensitive non-privileged instructions, we modify the guest kernel source code slightly. We do not have to worry about user space software as user space applications will execute in the same CPU mode as if they were executing directly on a physical machine. Sensitive instructions are not generated by standard C-compilers and are therefore only present in assembler files and inline assembly.

We modify the guest kernel source code using an automated scripting method. The script is based on regular expressions and has been tested on a number of kernel versions with success. The script supports inline assembler syntax, assembler as part of preprocessor macros, and, assembler macros.

It works by replacing sensitive non-privileged instructions with trap instructions and emulating the sensitive instruction in software when handling the trap. However, KVM/ARM must be able to retrieve the original sensitive instruction including its operands to be able to emulate the sensitive instruction when handling a trap. We experimented with inserting the trap instruction immediately *before* the sensitive instruction, but this caused problems with PC-relative addressing and *fix-up tables*.[4] To avoid the need to manually fix the patched code, we defined an encoding of all the sensitive non-privileged instructions and their operands into trap instructions.

The `SWI` instruction on ARM always traps and is normally used for making system calls. The instruction contains a 24-bit immediate field (the payload), which we can use to encode sensitive instructions. Unfortunately, the 24 bits are not quite enough to encode all the possible sensitive non-privileged instructions and their operands. However, all coprocessor access instructions trap if they access an undefined coprocessor. If we specify coprocessors zero through seven, which are not defined by the ARM architecture, all the coprocessor access functions will trap regardless of their operands. The coprocessor access instructions use 24 bits for their operands, which we can also leverage to encode the sensitive non-privileged instructions.

The VMM needs to be able to distinguish between guest system calls and traps for sensitive instructions. We make the assumption that the guest kernel does not make system calls to itself. Under this assumption, we simply interpret the payload if the virtual CPU is in privileged mode and emulate the encoded instruction. If the virtual CPU is in user mode, we consider the `SWI` instruction a system call made by guest user space to the guest kernel.

The ARM architecture defines 24 sensitive non-privileged instructions in total. We encode the instructions by grouping them in 15 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). This leaves us 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). We index them in two groups: one where the I bit is set and one where it's clear. In this way, the data processing instructions need only 16 bits to encode their operands

[4] Fix-up tables is a method used by the kernel to verify access on copy to and from user space operations. It uses offsets to the PC linked at a special section and added to the PC on memory access violations.

leaving us 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 12 of the remaining 13 index values as can be seen in Table 1.

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

Table 1: Sensitive instruction encoding types

In Table 1 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. We encode the versions with the I-bit set 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, we can use 2 bits to distinguish between the 4 sensitive load/store instructions. That gives us 22 bits to encode the instructions with the I-bit set, which is exactly what is needed.

We illustrate the implementation of our solution by an example. 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, we must make sure that it returns the virtual CPSR instead. Thus, we replace it 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/ARM loads the instruction from memory, decodes it, emulates it, and finally returns to line two.

The approach differs from Xen's paravirtualization solution in that it requires no knowledge of how the guest is engineered and can be applied automatically on any OS source tree compiled by GCC. For instance, Xen defines a whole new file in arch/arm/mm/pgtbl-xen.c, which contains functions based on other Xen macros to issue hypercalls regarding memory management. Calls to these functions are placed instead of existing kernel code through the use of preprocessor conditionals many places in the kernel code. The presented solution completely maintains the original kernel logic, which drastically reduces the engineering cost and makes the solution more suitable for test and development of existing kernel code.

3.2 Exceptions

An exception to an ARM processor is a common term for traps and interrupts. Traps are caused by software issuing an instruction that traps and interrupts are generated externally by hardware events. Common for all of them are that when they occur, the processor changes to a privileged mode and jumps to a predefined virtual memory address. The ARM architecture defines the exceptions shown in Table 2. It is configurable at run-time whether the low or high address shown in the table are used.

The reset exception occurs when the physical reset pin on the processor is asserted. Undefined exceptions happen when an unknown op-code is used for an instruction or when software in user mode try to access privileged coprocessor registers or when software access non-existing coprocessors. Software interrupt exceptions happen when SWI instructions are executed. Prefetch and data abort exceptions happen when the processor cannot fetch the next instruction or complete load/store instructions, respectively. These exceptions

Exception	Low addr.	High addr.
Reset	0x00000000	0xffff0000
Undefined	0x00000004	0xffff0004
Software interrupt	0x00000008	0xffff0008
Prefetch abort	0x0000000c	0xffff000c
Data abort	0x00000010	0xffff0010
Interrupt	0x00000018	0xffff0018
Fast-interrupt	0x0000001c	0xffff001c

Table 2: ARM exceptions overview

are either caused by missing page table entries or by memory protection violations. Interrupt and fast interrupts 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 have interrupts disabled at power on. During the boot process, the operating system makes sure not to generate any traps as these would cause unpredictable behavior. After the OS sets up page tables and enables virtual memory, it makes sure to map an *exception vector page* into 0xffff0000 (if high vectors are used). Only then it enables interrupts and starts generating traps. The instructions at the specific addresses in Table 2 are usually branch instructions to more or less complex handler functions.

When the guest runs, it is likely going to use almost the entire virtual address space. Especially if we are running the same guest and host OS, there is clearly going to be a conflict. Therefore, VMs execute in their own separate address space. The only host pages mapped in the VM address space are the exception vector page and the *shared page*. We explain the shared page in more details in Section 4.2.

When the CPU is executing guest code, the only way for KVM/ARM to regain control is through an exception. Unfortunately, we cannot use the host kernel exception handlers to handle exceptions when running the guest, but we have to write our own handlers. The KVM/ARM exception handlers are mapped at the exception vector page address in the VM's address space and are designed to re-enter the host kernel address space and return to host kernel code when an exception occurs. KVM/ARM then examines the guest exit reason and performs required emulation before resuming guest execution.

The above approach differs significantly from KVM on x86, which is based on hardware virtualization support. With hardware virtualization support, the exit path from guest execution is not through normal exceptions. Instead, the hardware automatically changes segment registers and thereby changes the address space back to the host kernel and resumes execution after the original world switch instruction. Software can then simply read a special register to determine the guest exit reason.

When KVM/ARM handles a hardware interrupt, it needs to run the host kernel hardware interrupt handler. If the host kernel handler is not run, the host kernel may miss important hardware events such as timer ticks or network packets. The host kernel interrupt handler queries the interrupt controller to find out what happened and manages the device that caused the interrupt as necessary. However, since we want to run the host kernel handler *after* the actual hardware interrupt went off, we are entering the host handler differently than usual. Unfortunately, it's not just a matter of a simple function call as the host kernel handler expects a certain state of the stack and CPU registers when the handler is called. KVM/ARM sets up a state exactly as it would have been if the host kernel interrupt handler had handled the interrupt directly, and executes the host kernel handler.

We note that in the typical use of KVM the guest kernel never needs to handle interrupts as a result of hardware interrupts. The guest kernel will only be exposed to emulated devices. Interrupts from emulated devices are generated by software and artificially injected into the guest. When this happens, or when we need to inject a trap to the guest (e.g. guest user space issues a system call to the guest kernel) an address space conflict can occur.

Suppose the guest is using the high address for the exception vector page and the host is also using the high address to handle actual hardware exceptions. The two physical frames can obviously not be mapped at the same virtual address during guest execution, and the KVM/ARM exception vector page must always be mapped, since otherwise there is no way for the host kernel to regain control of the system. When the guest kernel is about to handle an exception, it will trap on the instruction memory access permissions, since the guest does not have access to the KVM/ARM exception vector page. In this case we simply tell the hardware to use the low vector addresses and map the KVM/ARM exception page at that address instead. If later the guest needs to access the low vector address for other purposes, we simply tell the hardware to use the high vector addresses again. Switching the vector location is a very simple operation and involves only writing to a coprocessor register and invalidating a few TLB entries and cache lines.

The exception handlers used by KVM/ARM are compiled as a part of KVM and get linked at an unknown address. When a VM is created in KVM/ARM, the exception handlers are relocated from the address they were linked at to a newly allocated exception vector page belonging to that VM. Due to limited width of the immediate fields in ARM instructions even assembler code can generate binary code, which cannot simply be copied to a new location as it may reference data at specific addresses. Therefore the exception handler code is written in location-independent assembly. For instance, the following instruction would generate a load from a PC-relative address, which would not be easy to detect when relocating the code:

```
code_start:
   ldr     lr, =0xffff1000
   ...
code_end:
```

Instead, we write the code like the following, which loads the 32-bit immediate value from a local label relative to the PC. When we relocate the code segment we copy the code until the `code_end` label, which will include the data value:

```
code_start:
   ldr     lr, 1f
   ...
1: .word   0xffff1000
code_end:
```

4 Memory virtualization

Virtual machines need access to any part of the virtual address space they desire. But we cannot simply allow the guest OS to manage the physical memory or the MMU, as the host OS must retain control over physical memory and be protected from the guest at all times. Therefore, the memory must be virtualized.

The memory system on ARM exists in two flavors: one without an MMU (replaced by a *Memory Protection Unit* (MPU)) and one with an MMU. The latter translates virtual to physical addresses in hardware using a rather flexible two-level page table layout. The presented solution is developed for systems with an MMU. MMU-less cores are usually used in extremely simple embedded systems where virtualization may be of less importance anyway.

Memory virtualization introduces a new address space: *guest physical addresses*. Guest physical addresses are the addresses that the guest thinks represent physical memory addresses. However, since the memory is virtualized they are simply offsets into the memory region allocated to the guest. Guest page tables, which are managed by the guest kernel, translate from guest virtual addresses to guest physical addresses and can therefore not be used for address translation by the MMU. The guest physical addresses must first be translated to *host physical addresses* (also called machine addresses)[5]. See Figure 1 for an illustration of the address spaces.

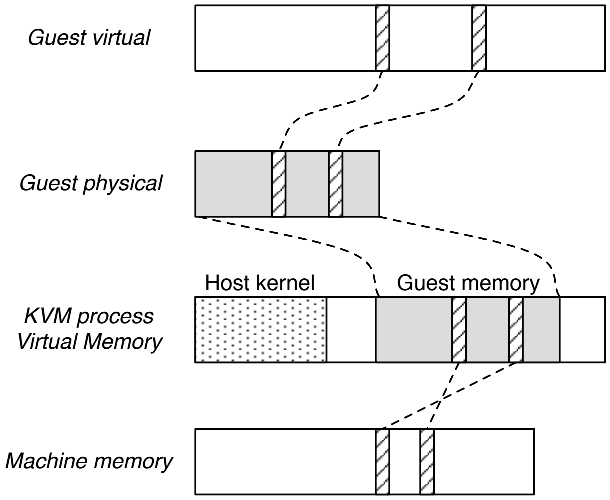

Figure 1: KVM address spaces

4.1 Shadow page tables

Shadow page tables are data structures managed by KVM/ARM, which are used by the hardware to translate from guest virtual addresses to machine addresses during guest execution.

[5] Recent hardware support for virtualization include technologies called Nested Page Tables or Extended Page Tables. These technologies add an extra translation table, which translates guest physical addresses to machine addresses in hardware.

When a new shadow page table is allocated, two special entries are always created and the rest of the table is left blank. The two special pages are the *shared page* explained in Section 4.2 and the exception vector page discussed in Section 3.2. If the guest tries to access any page other than the two just mentioned, a page fault occurs.

Guest pages are mapped into shadow page tables on demand, when KVM/ARM handles page faults occurring in VMs. The custom KVM/ARM exception handler will determine the virtual address which caused the fault and create an appropriate mapping in the shadow page table. Such a mapping must translate from the fault address to a machine address. KVM/ARM translates the fault address into a guest physical address by walking the guest page tables in software. The guest physical address is then translated to a host virtual address through architecture independent KVM functionality and finally the host virtual address is translated to a machine address by using standard kernel virtual memory translation functions.

ARM level-1 page table entries can be either *section descriptors*, pointers to *coarse page tables* or pointers to *fine page tables*. Coarse and fine page tables are commonly referred to as level-2 tables. Sections is a way to map a 1MB virtual memory region to a corresponding 1MB of contiguous physical memory. Coarse page table entries map pages of either 64KB (large pages), 4KB (small pages) or 1KB (tiny pages). Linux uses almost exclusively coarse page tables with pointers to 4KB small pages.

The entries in the shadow page tables should generally be of the same type as the entries in the guest page tables. However, this may not be possible when the guest uses section descriptors. Since Linux operates with a 4KB page size and KVM/ARM requests physical memory pages from Linux user pages, we cannot guarantee 1MB contiguous free physical memory on the host. Therefore, all shadow page table entries use 4KB small page mappings. This approach causes no loss in functionality, but it may affect performance negatively. The reason is that section descriptors only occupy a single entry in an ARM TLB, where a similar mapping of a 1MB area based on 4KB small pages occupy 256 TLB entries.

When the guest modifies page tables, KVM/ARM must also update the shadow page table. For instance, if the guest kernel uses copy-on-write, it will change the mapping on the first write to the COW section. Fortunately, the guest must invalidate the TLB when changing page tables. TLB invalidation is a privileged operation so KVM/ARM will catch the operation. When it happens, KVM/ARM simply re-initializes the shadow page table to only contain the shared page and the exception page and maps in the updated entries on demand.

4.2 Shared page and world switches

The shared page mentioned above is, as the name suggests, a page which is shared between the host and the guest. It is always mapped at the same address in the host kernel as in the guest kernel and it is used to perform world switches. The reason why we need a shared page is that guests execute in completely separate address spaces from the host and the ARM architecture requires that a change of address spaces is done from a page mapped at the same virtual address in both the previous and new address space - otherwise the behavior is unpredictable.

The shared page is mapped such that the guest cannot access the page and any reads or writes from or to the page will generate a page permission fault. If that happens, KVM/ARM must map the shared page at a different virtual address in both the host and the guest and map a guest page at the fault address instead. By doing so, the existence of the shared page is hidden from the guest OS. For Linux guests we take advantage of the reserved memory region in ARM Linux (see `Documentation/arm/memory.txt` for more info).

The shared page is structured as shown in Figure 2. The code contains two entry points: `__vcpu_run` and `__exception_return`. The first is used to switch to the guest and the second is called from the custom exception handlers to restore the host environment after an exception occurs. The data in the top of the page is needed to complete world switches. For instance, when switching to the guest, the code in the shared page reads the base physical address of the active level-1 page table from a special register called the *Translation Table Base Register* (TTBR) and stores the value at the top of the shared page. The data section also contains the physical base address of the shadow page tables and the world switch code writes this value to the hardware TTBR. After the page tables have been switched, the stack pointer is no longer valid as the kernel stack location is no longer mapped to the right physical memory. Therefore we reserve 2K from the top of the shared page for the stack.

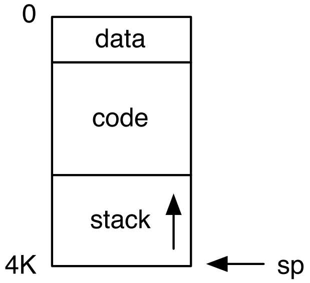

Figure 2: Shared page layout

Because KVM is compiled and distributed as a module, and because we want to be able to run multiple guests simultaneously[6], and because we may not know the shared virtual address at compile time, the code on the shared page can not be linked directly by the kernel. Instead, a shared page is allocated per VM, and the world switch code is relocated to the corresponding shared page. Like the code for the exception vector page, the code for the shared page is written in position-independent assembly to make it relocatable.

4.3 Memory protection

Memory protection on ARM works through two concepts: Domains and Access Permissions. These features are used by traditional ARM operating systems to protect for instance kernel memory from user space applications and to implement features such as copy-on-write.

There are 16 domains on ARM - domain 0 to 15. Each level-1 page table entry describes a 1MB virtual address region. The level-1 entries contain a 4-bit value denoting which domain the 1MB address region belongs to. Each domain can be in one of the following three modes:

- No access
- Client
- Manager

If a page belongs to the no access domain, all accesses to that page will generate faults. If a page belongs to an administrator domain, all accesses will succeed. Finally, if a page belongs to a client domain, the access permissions on page table entries are checked.

The access permissions define permissions depending on the privilege level of the CPU. In Table 3 we show the possible access permissions for normal memory on ARMv5.

AP[1:0]	Privileged	User
0b00	No access	No access
0b01	Read/write	No access
0b10	Read/write	Read only
0b11	Read/write	Read/write

Table 3: Access permission settings

Since the guest will always run in user mode, KVM/ARM must translate the guest page access permissions to a value resulting in the same level of protection on the shadow page tables. For example, if the guest page tables allow read/write access in privileged mode but no access in user mode, and the VM is in privileged mode, the shadow page table access permissions must use read/write for user mode. When the VM changes CPU mode, KVM/ARM updates access permissions on shadow page table entries correspondingly. See Table 4 for the translation scheme used to translate between guest access permissions and shadow access permissions.

Domains are essentially easy to handle in shadow page tables, since their settings can simply be copied from the guest page tables and used on the shadow page tables. However, the shared page and the exception vector page must always have read/write privileged access and no user mode access, in order to protect the host from the guest. Consequently these pages must be mapped using a client domain. Since the domain is specified for a 1MB virtual address space range, a single level-1 page table entry can specify the domain for both pages belonging to the guest and for the exception vector page or the shared page or both. In this case, KVM/ARM changes the level-1 page table entry domain to a client domain and modifies access permissions on the shadow page table entries in the same 1MB address range to correspond to the guest domain setting.

[6] Since the data on the shared page belongs to the specific VM, the shared page cannot be shared across VMs

	Guest page table permissions			Shadow page table permissions		
Guest mode	Priv.	User	AP	Priv.	User	AP
User	NA	NA	00	RW	NA	01
Priv.					NA	01
User	RW	NA	01	RW	NA	01
Priv.					RW	11
User	RW	RO	10	RW	RO	10
Priv.					RW	11
User	RW	RW	11	RW	RW	11
Priv.					RW	11

Table 4: Access permission translation from guest to shadow page tables

5 Implementation status

The presented work is based on the 2.6.27 kernel running on the Google Android emulator, which emulates the ARMv5 architecture (specifically an arm926E core) on a custom "Goldfish" platform. The emulator provided us with a very convenient development environment as it supports GDB debugging of kernel code.

The solution successfully boots a Linux kernel with a simple user space init environment. We can run small programs although with poor performance. This was not surprising, as the implementation has not been optimized for performance. Instead, the work has been focused on correct functionality and full support for all architecture features. As a consequence many features have been implemented naively, which eased debugging and code clarity and performance optimizations have been postponed for future work.

The VM supports the devices on a standard ARM integrator development platform. Theoretically, since all devices are memory-mapped on ARM and thereby communicate with QEMU using the same functionality, all devices should work once a single device works. However, there may be timing constraints which requires the use of paravirtualized drivers, coalesced MMIO or something completely different.

The MMU emulation does not support tiny pages. The reason is simply that we have not come across a guest using them yet. Further, the use of tiny pages is deprecated from ARMv6 and forward.

We are currently working on ARMv6 and ARMv7 support. The ARMv6 work is done on the HTC Dream G1 developer phone which is equipped with a ARM1136EJ-S core. The ARMv7 work is done on BeagleBoards which feature Cortex-A8 cores. The majority of this work consists of supporting the new page table formats introduced in ARMv6, supporting processor-specific cache and TLB manipulation functions, ensuring cache coherency on shared data, and adding emulation code to support a few instructions added in ARMv6 such as `SRS` and `RFE`.

Additionally we are also taking steps to improve the performance of the system. Specifically, we are taking advantage of new ARMv6 features to reduce the world switch costs. ARMv6 and newer supports physically tagged caches, which in part avoids the need to flush caches on world switches. Further, ARMv6 supports tagging of TLB entries with *Application Space Identifiers* (ASIDs), which allows several TLB entries with the same virtual address, but belonging to different address spaces, to reside in the TLB at the same time. By using ASIDs we can avoid TLB invalidations on world switches. Finally, we are also experimenting with other performance improvements such as removing unnecessary memcopies, caching shadow page tables and more.

ARM TrustZone[4] is a security technology available in a number of recent processors from ARM and is the closest thing to hardware support for virtualization on ARM. In contrast to the latest versions of x86 hardware virtualization extensions, TrustZone does not support memory virtualization and TrustZone does not provide functionality making it easier to decode sensitive instructions for emulation. We plan to further investigate the benefits and options for using TrustZone with KVM/ARM. PikeOS by SYSGO is an ARM hypervisor targeted towards security critical real-time systems.

PikeOS is based on TrustZone [7] and thereby limited to processors with programmable TrustZone support. Unfortunately there are no technical details available on how their solution is engineered in details.

We hope to get the solution included as part of the mainline kernel and QEMU source trees. The solution hardly modifies any code outside the KVM module and even inside KVM there are very few changes to the architecture independent code.

6 Conclusions

We have presented KVM/ARM, the first working open-source ARM virtualization solution based on KVM. Although ARM is not virtualizable, we show how lightweight paravirtualization can be used to automatically enable commodity operating systems to run in a KVM/ARM virtual machine. The approach is minimally intrusive and requires no knowledge or understanding of commodity operating system code, making it relatively easy for KVM/ARM to support more recent versions of commodity operating systems. By building on KVM, KVM/ARM can enjoy the same benefits of KVM, including having its code base kept in line with new kernel releases without additional maintenance costs and being easily included in most Linux distributions. We have implemented a KVM/ARM prototype based on the Linux kernel used in Google Android and have demonstrated its ability to successfully run nearly unmodified Linux guest operating systems.

Our KVM/ARM implementation work has benefited from community support and improvements are ongoing. Newer versions of the ARM architecture provide features to improve KVM/ARM and bring virtualization to embedded devices, smartbooks, and mobile phones in the future. For more information about the project or to get involved, please refer to the project wiki at: `https://wiki.ncl.cs.columbia.edu/wiki/index.php/AndroidVirt:MainPage`.

7 Acknowledgments

Hollis Blanchard, Alexander Graf, and Oren Laadan provided many helpful suggestions in implementing and debugging KVM/ARM. This work was supported in part by NSF grants CNS-0914845 and CNS-0905246, and AFOSR MURI grant FA9550-07-1-0527.

References

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

[2] AMD Virtualization (AMD-V^{TM}) Technology. `http://sites.amd.com/us/business/it-solutions/virtualization/Pages/amd-%v.aspx`.

[3] ARM Ltd. ARM Architecture Reference Manual (ARM DDI 0100I), 2005.

[4] ARM Ltd. TrustZone Security White Paper, 2010. `http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD%29-GENC-009492C_trustzone_security_whitepaper.pdf`.

[5] 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 *SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[6] Robert P. Goldberg. A Survey of Virtual Machine Research. *IEEE Computer*, 7(6):34–45, 1974.

[7] Green Hills Software Inc. White paper: Integrity Secure Virtualization for ARM, 2010. `http://www.ghs.com/ds/index.php?ds=integrity_virt_ARM`.

[8] J-Y. Hwang, S-B. Suh, S-K. Heo, C-J. Park, J-M. Ryu, S-Y. Park, and C-R. Kim. Xen on ARM: System Virtualization using Xen hypervisor for ARM-based Secure Mobile Phones. In *Fifth IEEE Consumer Communications and Networking Conference*, pages 257–261, 2008.

[9] Intel®Virtualization Technology (Intel®VT). `http://www.intel.com/technology/virtualization/technology.htm`.

[10] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. **kvm**: The Linux Virtual

Machine Monitor. In *OLS 2007: Proceedings of the Linux Symposium*, pages 225–230, 2007.

[11] Open Kernel Labs. OKL4 Microvisor. `http://www.ok-labs.com/products/okl4-microvisor`.

[12] Gerald J. Popek and Robert P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[13] Mendel Rosenblum and Tal Garfinkel. Virtual Machine Monitors: Current Technology and Future Trends. *IEEE Computer*, pages 39–47, 2005.

[14] Sang-bum Suh. Presentation: Secure Architecture and Implementation of Xen on ARM for Mobile Devices, 2007. `http://www.xen.org/files/xensummit_4/Secure_Xen_ARM_xen-summit-04_07_Su%h.pdf`.

[15] VirtualLogix. Real-time Virtualization for Conencted Devices. `http://www.virtuallogix.com/solutions/product/arm.html`.

[16] VMware. VMware Mobile Virtualization Platform, Virtual Appliances for Mobile phones. `http://www.vmware.com/products/mobile/`.

[17] 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.

Proceedings of the Linux Symposium

July 13th–16th, 2010
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium,*
Thin Lines Mountaineering

Programme Committee

Andrew J. Hutton, *Linux Symposium*
Martin Bligh, *Google*
James Bottomley, *Novell*
Dave Jones, *Red Hat*
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Matthew Wilson

Proceedings Committee

Robyn Bergeron

With thanks to
John W. Lockhart, *Red Hat*