# Supporting KVM on the ARM architecture

**July 3, 2013**
This article was contributed by Christoffer Dall and Jason Nieh

One of the new features in the 3.9 kernel is KVM/ARM: KVM support for the ARM architecture. While KVM is already supported on i386 and x86/64, PowerPC, and s390, ARM support required more than just reimplementing the features and styles of the other architectures. The reason is that the ARM virtualization extensions are quite different from those of other architectures.

Historically, the ARM architecture is not virtualizable, because there are a number of sensitive instructions which do not trap when they are executed in an unprivileged mode. However, the most recent 32-bit ARM processors, like the Cortex-A15, include hardware support for virtualization as an ARMv7 architectural extension. A number of research projects have attempted to support virtualization on ARM processors without hardware virtualization support, but they require various levels of paravirtualization and have not been stabilized. KVM/ARM is designed specifically to work on ARM processors with the virtualization extensions enabled to run unmodified guest operating systems.

The ARM hardware extensions differ quite a bit from their x86 counterparts. A simplified view of the ARM CPU modes is that the kernel runs in SVC mode and user space runs in USR mode. ARM introduced a new CPU mode for running hypervisors called HYP mode, which is a more privileged mode than SVC mode. An important characteristic of HYP mode, which is central to the design of KVM/ARM, is that HYP mode is not an extension of SVC mode, but a distinct mode with a separate feature set and a separate virtual memory translation mechanism. For example, if a page fault is taken in HYP mode, the faulting virtual address is stored in a different register in HYP mode than in SVC mode. As another example, for the SVC and USR modes, the hardware has two separate page table base registers, which are used to provide the familiar address space split between user space and kernel. HYP mode only uses a single page table base register and therefore does not allow the address space split between user mode and kernel.

The design of HYP mode is a good fit with a classic bare-metal hypervisor design because such a hypervisor does not reuse any existing kernel code written to work in SVC mode. KVM, however, was designed specifically to reuse existing kernel components and integrate these with the hypervisor. In comparison, the x86 hardware support for virtualization does not provide a new CPU mode, but provides an orthogonal concept known as "root" and "non-root". When running as non-root on x86, the feature set is completely equivalent to a CPU without virtualization support. When running as root on x86, the feature set is *extended* to add additional features for controlling virtual machines (VMs), but all existing kernel code can run unmodified as both root and non-root. On x86, when a VM traps to the hypervisor, the CPU changes from non-root to root. On ARM, when a VM traps to the hypervisor, the CPU traps to HYP mode.

HYP mode controls virtualization features by configuring sensitive operations to trap to HYP mode when executed in SVC and USR mode; it also allows hypervisors to configure a number of shadow register values used to hide information about the physical hardware from VMs. HYP mode also controls Stage-2 translation, a feature similar to Intel's "extended page table" used to control VM memory access. Normally when an ARM processor issues a load/store instruction, the memory address used in the instruction is translated by the memory management unit (MMU) from a virtual address to a physical address using regular page tables, like this:

- Virtual Address (VA) -> Intermediate Physical Address (IPA)

The virtualization extensions add an extra stage of translation known as Stage-2 translation which can be enabled and disabled only from HYP mode. When Stage-2 translation is enabled, the MMU translates address in the following way:

- Stage-1: Virtual Address (VA) -> Intermediate Physical Address (IPA)
- Stage-2: Intermediate Physical Address (IPA) -> Physical Address (PA)

The guest operating system controls the Stage-1 translation independently of the hypervisor and can change mappings and page tables without trapping to the hypervisor. The Stage-2 translation is controlled by the hypervisor, and a separate Stage-2 page table base register is accessible only from HYP mode. The use of Stage-2 translations allows software running in HYP mode to control access to physical memory in a manner completely transparent to a VM running in SVC or USR mode, because the VM can only access pages that the hypervisor has mapped from an IPA to the page's PA in the Stage-2 page tables.

## KVM/ARM design

KVM/ARM is tightly integrated with the kernel and effectively turns the kernel into a first class ARM hypervisor. For KVM/ARM to use the hardware features, the kernel must somehow be able to run code in HYP mode because HYP mode is used to configure the hardware for running a VM, and traps from the VM to the host (KVM/ARM) are taken to HYP mode.

Rewriting the entire kernel to run only in HYP mode is not an option, because it would break compatibility with hardware that doesn't have the virtualization extensions. A HYP-mode-only kernel also would not work when run inside a VM, because the HYP mode would not be available. Support for running both in HYP mode and SVC mode would be much too invasive to the source code, and would potentially slow down critical paths. Additionally, the hardware requirements for the page table layout in HYP mode are different from those in SVC mode in that they mandate the use of LPAE (ARM's Large Physical Address Extension) and require specific bits to be set on the page table entries, which are otherwise clear on the kernel page tables used in SVC mode. So KVM/ARM must manage a separate set of HYP mode page tables and explicitly map in code and data accessed from HYP mode.

We therefore came up with the idea to split execution across multiple CPU modes and run as little code as possible in HYP mode. The code run in HYP mode is limited to a few hundred instructions and isolated to two assembly files: `arch/arm/kvm/interrupts.S` and `arch/arm/kvm/interrupts_head.S`.

For readers not familiar with the general KVM architecture, KVM on all architectures works by exposing a simple interface to user space to provide virtualization of core components such as the CPU and memory. Device emulation, along with setup and configuration of VMs, is handled by a user space process, typically QEMU. When such a process decides it is time to run the VM, it will call the `KVM_VCPU_RUN` `ioctl()`, which executes VM code natively on the CPU. On ARM, the `ioctl()` handler in `arch/arm/kvm/arm.c` switches to HYP mode by issuing an HVC (hypercall) instruction, which changes the CPU mode to HYP mode, context switches all hardware state between the host and the guest, and finally jumps to the VM SVC or USR mode to natively execute guest code. When KVM/ARM runs guest code, it enables Stage-2 memory translation, which completely isolates the address space of VMs from the host and other VMs. The CPU will be executing guest code until the hardware traps to HYP mode, because of a hardware interrupt, a stage-2 page fault, or a sensitive operation. When such a trap occurs, KVM/ARM switches back to the host hardware state and returns to normal KVM/ARM host SVC code with the full kernel mappings available.

When returning from a VM, KVM/ARM examines the reason for the trap, and performs the necessary emulation or resource allocation to allow the VM to resume. For example, if the guest performs a memory-mapped I/O (MMIO) operation to an emulated device, that will generate a Stage-2 page fault, because only physical RAM dedicated to the guest will be mapped in the Stage-2 page tables. KVM/ARM will read special system registers, available only in HYP mode, which contain the address causing the fault and report the address to QEMU through a shared memory-mapped structure between QEMU and the kernel. QEMU knows the memory map of the emulated system and can forward the operation to the appropriate device emulation code. As another example, if a hardware interrupt occurs while the VM is executing, this will trap to HYP mode, and KVM/ARM will switch back in the host state and re-enable interrupts, which will cause the hardware interrupt handlers to execute once again, but this time without trapping to HYP mode. While every hardware interrupt ends up interrupting the CPU twice, the actual trap cost on ARM hardware is negligible compared to the world-switch from the VM to the host.

## HYP mode

Providing access to HYP mode from KVM/ARM was a non-trivial challenge, since HYP mode is a more privileged mode than the standard ARM kernel modes and there is no architecturally defined ABI for entering HYP mode from less privileged modes. One option would be to expect bootloaders to either install secure monitor handlers or hypercall handlers that would allow the kernel to trap back into HYP mode, but this method is brittle and error-prone, and prior experience with establishing TrustZone APIs has shown that it is hard to create a standard across different implementations of the ARM architecture.

Instead, Will Deacon, Catalin Marinas, and Ian Jackson proposed that we rely on the kernel being booted in HYP mode if the kernel is going to support KVM/ARM. In version 3.6, a patch series developed by Dave Martin and Marc Zyngier was merged that detects if the kernel is booted in HYP mode and, if so, installs a small stub handler that allows other subsystems like KVM/ARM to take control of HYP mode later on. As it turns out, it is reasonable to recommend that bootloaders always boot the kernel in HYP mode if it is available because even legacy kernels always make an explicit switch to SVC mode at boot time, even though they expect to boot into SVC mode already. Changing bootloaders to simply boot all kernels in HYP mode is therefore backward-compatible with legacy kernels.

Installing the hypervisor stub when the kernel is booted in HYP mode was an interesting implementation challenge. First, ARM kernels are often loaded as a compressed image, with a small uncompressed pre-boot environment known as the "decompressor" which decompresses the kernel image into memory. If the decompressor detects that it is booted in HYP mode, then a temporary stub must be installed at this stage allowing the CPU to fall back to SVC mode to run the decompressor code. The reason is that the decompressor must turn on the MMU to enable caches, but doing so in HYP mode requires support for the LPAE page table format used by HYP mode, which is an unwanted piece of complexity in the decompressor code. Therefore, the decompressor installs the temporary HYP stub, falls back to SVC mode, decompresses the kernel image, and finally, immediately before calling the uncompressed initialization code, switches back to HYP mode again. Then, the uncompressed initialization code will again detect that the CPU is in HYP mode and will install the main HYP stub to be used by kernel modules later in the boot process or after the kernel has finally booted. The HYP stub can be found in `arch/arm/kernel/hyp-stub.S`. Note that the uncompressed initialization code doesn't care whether the uncompressed code is started directly in HYP mode from a bootloader or from the decompressor.

Because HYP mode is a more privileged mode than SVC mode, the transition from SVC mode to HYP mode occurs only through a hardware trap. Such a trap can be generated by executing the hypercall (HVC) instruction, which will trap into HYP mode and cause the CPU to execute code from a jump entry in the HYP exception vectors. This allows a subsystem to use the hypervisor stub to fully take over control of HYP mode, because the hypervisor stub allows subsystems to change the location of the exception vectors. The HYP stub is called through the `__hyp_set_vectors()` function, which takes the physical address of the HYP exception vector as its only parameter, and replaces the HYP Vector Base Address Register (HVBAR) with that address. When KVM/ARM is initialized during normal kernel boot (after all main kernel initialization functions have run), it creates an identity mapping (one-to-one mapping of virtual addresses to physical address) of the HYP mode initialization code, which includes an exception vector, and sets the physical address of using the `__hyp_set_vectors()` function. Further, the KVM/ARM initialization code calls the HVC instruction to run the identity-mapped initialization code, which can safely enable the MMU, because the code is identity mapped.

Finally, KVM/ARM initialization sets up the HVBAR to point to the main KVM/ARM HYP exception handling code, now using the virtual addresses for HYP mode. Since HYP mode has its own address space, KVM/ARM must choose an appropriate virtual address for any code or data, which is mapped into HYP mode. For convenience and clarity, the kernel virtual addresses are reused for pages mapped into HYP mode, making it possible to dereference structure members directly as long as all relevant data structures are mapped into HYP mode.

Both traps from sensitive operations in VMs and hypercalls from the host kernel enter HYP mode through an exception on the CPU. Instead of changing the HYP exception vector on every switch between the host and the guest, a single HYP exception vector is used to handle both HVC calls from the host kernel and to handle traps from the VM. The HYP vector handling code checks the VMID field on the Stage-2 page table base register, and VMID 0 is reserved for the host. This field is only accessible from HYP mode and guests are therefore prevented from escalating privilege. We introduced the `kvm_call_hyp()` function, which can be used to execute code in HYP mode from KVM/ARM. For example, KVM/ARM code running in SVC mode can make the following call to invalidate TLB entries, which must be done from HYP mode:

```
    kvm_call_hyp(__kvm_tlb_flush_vmid_ipa, kvm, ipa);
```

## Virtual GIC and timers

ARMv7 architectures with hardware virtualization support also include virtualization support for timers and the interrupt controller. Marc Zyngier implemented support for these features, which are called "generic timers" (a.k.a. architected timers) and the Virtual Generic Interrupt Controller (VGIC).

Traditionally, timer operations on ARM systems have been MMIO operations to dedicated timer devices. Such MMIO operations performed by VMs would trap to QEMU, which would involve a world-switch from the VM to host kernel, and a switch from the host kernel to user space for every read of the time counter or every time a timer needed to be programmed. Of course, the timer functionality could be emulated inside the kernel, but this would require a trap from the VM to the host kernel, and would therefore add substantial overhead to VMs compared to running on native hardware. Reading the

counter is a very frequent operation in Linux. For example, every time a task is enqueued or dequeued in the scheduler, the runqueue clock is updated, and in particular multi-process workloads like Apache benchmarks clearly show the overhead of trapping on each counter read.

ARMv7 allows for an optional extension to the architecture, the generic timers, which makes counter and timer operations part of the core architecture. Now, reading a counter or programming a timer is done using coprocessor register accesses on the core itself, and the generic timers provide two sets of timers and counters: the physical and the virtual. The virtual counter and timer are always available, but access to the physical counter and timer can be limited through control registers accessible only in HYP mode. If the kernel is booted in HYP mode, it is configured to use the physical timers; otherwise the kernel uses the virtual timers, allowing both an unmodified kernel to program timers when running inside a VM without trapping to the host, and providing the necessary isolation of the host from VMs.

If a VM programs a virtual timer, but is preempted before the virtual timer fires, KVM/ARM reads the timer settings to figure out the remaining time on the timer, and programs a corresponding soft timer in the kernel. When the soft timer expires, the timer handler routine injects the timer interrupt back into the VM. If the VM is scheduled before the soft timer expires, the virtual timer hardware is re-programmed to fire when the VM is running.

The role of an interrupt controller is to receive interrupts from devices and forward them to one or more CPUs. ARM's Generic Interrupt Controller (GIC) provides a "distributor" which is the core logic of the GIC and several CPU interfaces. The GIC allows CPUs to mask certain interrupts, assign priority, or set affinity for certain interrupts to certain CPUs. Finally, a CPU also uses the GIC to send inter-processor interrupts (IPIs) from one CPU core to another and is the underlying mechanism for SMP cross calls on ARM.

Typically, when the GIC raises an interrupt to a CPU, the CPU will acknowledge the interrupt to the GIC, interact with the interrupting device, signal end-of-interrupt (EOI) to the GIC, and resume normal operation. Both acknowledging and EOI-signaling interrupts are privileged operations that will trap when executed from within a VM, adding performance overhead to common operations. The hardware support for virtualization in the VGIC comes in the form of a virtual CPU interface that CPUs can query to acknowledge and EOI virtual interrupts without trapping to the host. The hardware support further provides a virtual control interface to the VGIC, which is accessed only by KVM/ARM, and is used to program virtual interrupts generated from virtual devices (typically emulated by QEMU) to the VGIC.

Since access to the distributor is typically not a common operation, the hardware does not provide a virtual distributor, so KVM/ARM provides in-kernel GIC distributor emulation code as part of the support for VGIC. The result is that VMs can acknowledge and EOI virtual interrupts directly without trapping to the host. Actual hardware interrupts received during VM execution always trap to HYP mode, and KVM/ARM lets the kernel's standard ISRs handle the interrupt as usual, so the host remains in complete control of the physical hardware.

There is no mechanism in the VGIC or generic timers to let the hardware directly inject physical interrupts from the virtual timers as virtual interrupts to the VMs. Therefore, VM timer interrupts will trap as any other hardware interrupt, and KVM/ARM registers a handler for the virtual timer interrupt and injects a corresponding virtual timer interrupt using software when the handler function is called from the ISR.

## Results

During the development of KVM/ARM, we continuously measured the virtualization overhead and ran long-running workloads to test stability and measure performance. We have used various kernel

configurations and user space environments (both ARM and Thumb-2) for both the host and the guest, and validated our workloads with SMP and UP guests. Some workloads have run for several weeks at a time without crashing, and the system behaves as expected when exposed to extreme memory pressure or CPU over-subscription. We therefore feel that the implementation is stable and encourage users to try and use the system.

Our measurements using both micro and macro benchmarks show that the overhead of KVM/ARM is within 10% of native performance on multicore platforms for balanced workloads. Purely CPU-bound workloads perform almost at native speed. The relative overhead of KVM/ARM is comparable to KVM on x86. For some macro workloads, like Apache and MySQL, KVM/ARM even has less overhead than on x86 using the same configuration. A significant source of this improved performance can be attributed to the optimized path for IPIs and thereby process rescheduling caused by the VGIC and generic timers hardware support.

**Status and future work**

KVM/ARM started as a research project at Columbia University and was later supported by Virtual Open Systems. After the 3.9 merge, KVM/ARM continues to be maintained by the original author of the code, Christoffer Dall, and the ARMv8 (64-bit) port is maintained by Marc Zyngier. QEMU system support for ARMv7 has been merged upstream in QEMU, and kvmtool also has support for KVM/ARM on both ARMv7 and ARMv8. ARMv8 support is scheduled to be merged for the 3.11 kernel release.

Linaro is supporting a number of efforts to make KVM/ARM itself feature complete, which involves debugging and full migration features including migration of the in-kernel support for the VGIC and the generic timers. Additionally, virtio has so far relied on a PCI backend in QEMU and the kernel, but a significant amount of work has already been merged upstream to refactor the QEMU source code concerning virtio to allow better support for MMIO-based virtio devices to accelerate virtual network and block devices. The remaining work is currently a priority for Linaro, as is support for the mach-virt ARM machine definition, which is a simple machine model designed to be used for virtual machines and is based only on virtio devices. Finally, Linaro is also working on ARMv8 support in QEMU, which will also take advantage of mach-virt and virtio support.

**Conclusion**

KVM/ARM is already used heavily in production by the SUSE Open Build Service on Arndale boards, and we can only speculate about its future uses in the green data center of the future, as the hypervisor of choice for ARM-based networking equipment, or even ARM-based laptops and desktops.

For more information, help on how to run KVM/ARM on your specific board or SoC, or to participate in KVM/ARM development, the kvmarm mailing list is a good place to start.

---

(Log in to post comments)

**Supporting KVM on the ARM architecture**

Posted Jul 4, 2013 1:14 UTC (Thu) by **smoogen** (subscriber, #97) [Link]

Here is the "Shut up and take my money" question :)

What hardware can I buy that supports this and where can I get it?

**Supporting KVM on the ARM architecture**

Posted Jul 4, 2013 4:42 UTC (Thu) by **olof** (subscriber, #11729) [Link] http://howchip.com/shop/content.php?co_id=ArndaleBoard_en for the Arndale development board.

The Samsung ARM-based Chromebook has the same SoC in it, but unfortunately the firmware on the system does not launch the kernel in HYP mode, so you can't run KVM out of the box on them (I.e. not even with the chainloaded nv-u-boot).

Some of the lower-end Cortex-A7 based systems will be able to run KVM too, for example Cubieboard 2 (which has the dual-A7 Allwinner A20 SoC on it). Upstream support for their hardware is not quite there yet though, but down the road things should look quite a bit better.

**Supporting KVM on the ARM architecture**

Posted Jul 4, 2013 13:59 UTC (Thu) by **ibukanov** (subscriber, #3942) [Link]

On ARM Chromebook is that ROM firmware that http://www.chromium.org/chromium-os/developer-information... mention that does not activate HYP?

If so one need to apply some soldering skills to replace it...

**Supporting KVM on the ARM architecture**

Posted Jul 5, 2013 11:01 UTC (Fri) by **danpb** (subscriber, #4831) [Link]

> The Samsung ARM-based Chromebook has the same SoC in it, but
> unfortunately the firmware on the system does not launch the
> kernel in HYP mode, so you can't run KVM out of the box on
> them (I.e. not even with the chainloaded nv-u-boot).

Hmm, this page claims to have tested KVM on the ARM Chromebook

http://columbia.github.io/linux-kvm-arm/

but they don't mention what they did about the bootloader/HYP mode problem. The Xen folks seem to suggest it is just a matter of compiling a version of u-boot with HYP mode support, and presumably they then chainloaded it

http://blog.xen.org/index.php/2013/05/23/bringing-xen-on-...

**Supporting KVM on the ARM architecture**

Posted Jul 5, 2013 23:32 UTC (Fri) by **christofferdall** (subscriber, #63430) [Link]

We have heard from a number of people who have successfully booted KVM on the chromebook using the HYP U-boot approach, but I haven't verified this myself.

However, given that you can chainload u-boot and that the device boots in secure mode it should be possible with enough effort to run KVM on there.

**Supporting KVM on the ARM architecture**

Posted Jul 4, 2013 9:37 UTC (Thu) by **bboy** (subscriber, #63074) [Link]

Arndale board or big.LITTLE TC2 platform are the right choices.

**Supporting KVM on the ARM architecture**

Posted Jul 9, 2013 13:40 UTC (Tue) by **jzbiciak** (subscriber, #5246) [Link]

This is an excellent, excellent article. I've been studying ARM's virtualization hardware support, and wondered just exactly how Linux would make use of it, due to all the differences you described in your opening paragraphs. You've answered those questions thoroughly!

Basically, as I understand the concept, the host Linux runs at SVC level like any other guest, but, it has a special channel to communicate with a thin layer at HYP level to control all the HYP-specific machinery to control the other guests. Did I understand that correctly?

One very minor quibble. When describing address translation, you say:

> *Normally when an ARM processor issues a load/store instruction, the memory address used in the instruction is translated by the memory management unit (MMU) from a virtual address to a physical address using regular page tables, like this:*
>
> > *Virtual Address (VA) -> Intermediate Physical Address (IPA)*

Shouldn't that just be "Virtual Address (VA) -> Physical Address (PA)"? I didn't think there was a notion of IPA when Stage-2 is not enabled. As I said, a very minor quibble.

### Supporting KVM on the ARM architecture

Posted Jul 9, 2013 15:15 UTC (Tue) by **raven667** (subscriber, #5198) [Link]

> Basically, as I understand the concept, the host Linux runs at SVC level like any other guest, but, it has a special channel to communicate with a thin layer at HYP level to control all the HYP-specific machinery to control the other guests. Did I understand that correctly?

That sounds like how every other hypervisor such as Xen or VMware is architected. Xen has dom0 and VMware has the Service Console (or whatever it's called on ESXi). This seems to be a pattern in the design.

### Supporting KVM on the ARM architecture

Posted Jul 11, 2013 6:44 UTC (Thu) by **christofferdall** (subscriber, #63430) [Link]

It's not exactly how other hypervisors work.

Xen's dom0 for example does not handle scheduling and memory allocation of VMs and boots directly into the Xen hypervisor in HYP mode and runs dom0 in many ways just like another guest. KVM/ARM boots in HYP, but immediately drops back into SVC mode, only returning to HYP mode to run a minimal amount of code in the most privileged mode.

VMware ESX and KVM on x86 both benefit from the x86 architecture, where there is no equivalent to the distinction between HYP mode and SVC mode on the host side - there is simply "root ring0" privilege level, and don't have to deal with separate mappings of code between the host kernel and the low-level part of the hypervisor, just to give an example.

In fact, this was the real challenge with designing KVM/ARM, that the architecture doesn't apparently fit very well with a KVM architecture, but our results show that this is not a significant concern for performance, and the it works out quite well from a software engineering point of view.

### Supporting KVM on the ARM architecture

Posted Jul 11, 2013 6:36 UTC (Thu) by **christofferdall** (subscriber, #63430) [Link]

yes, when Stage-2 translation is not used, the IPA equals the PA.

### Supporting KVM on the ARM architecture

Posted Dec 5, 2014 18:22 UTC (Fri) by **jcm** (subscriber, #18262) [Link]

Note that a couple of issues raised in this piece have been addressed as part of ARMv8.1.

### Supporting KVM on the ARM architecture

Posted Dec 24, 2016 10:51 UTC (Sat) by **raziel** (subscriber, #113201) [Link]

I am reading the code of the KVM/ARM64 and I am trying to understand whether it is possible to leave the hcr_el2 and the vttbr_el2 registers enabled without running any code in the guest at all.
I just want to monitor interrupts.

Whenever I do that the OS hangs due to a failure in the el1_irq even though I do not pass the interrupt to a guest.