Virtualization in the ARM Architecture
Lecture for the Embedded Systems Course
CSD, University of Crete (May 6 & 8, 2019)

- Manolis Marazakis (maraz@ics.forth.gr)

Institute of Computer Science (ICS)
Foundation for Research and Technology – Hellas (FORTH)
Virtualization Use-cases

- Enterprise
- Consumer Electronics
- Industrial
- Automotive
Virtualization Benefits in Embedded Systems

- Workload consolidation
  - E.g. Applications + Baseband sharing a multicore SoC
- Flexible resource provisioning
- License barrier
- Legacy software support
  - important with the multitude and variety of embedded operating systems (commercial and even home-brew)
- Reliability
- Security
Virtualization trade-off

- **Performance:**
  - Applications that used to own the whole processor must now share
  - Hypervisor adds runtime overhead & increases memory footprint
    - Real-time properties?
  - Full virtualization without hardware support means software emulation

- **Complexity:**
  - Old scenario: two software stacks + two hardware systems
  - New scenario: two software stacks + one hardware system + one host kernel
  - More abstraction layers → more bugs …

- **Security & reliability:**
  - Increased size of Trusted Computing Base (TCB)
  - Increased impact of hardware failure

- **I/O:** emulation vs (para)virtual vs direct access
Essentials of a hypervisor

- Parent partition (minimum-footprint OS) + Hypervisor
- **Hypervisor**: Thin layer of software running on the hardware
  - Supports creation of *partitions (virtual machines)*
    - Each partition has one or more *virtual processors*
    - Partitions can own or share hardware resources
- **Enforces memory access rules**
- **Enforces policy for CPU usage**
  - Virtual processors are scheduled on real processors
- **Enforces ownership of other devices**
- **Provides inter-partition messaging**
  - Messages appear as interrupts
- **Exposes simple programmatic interface: “hypercalls”**
Hypervisor functions

- Memory management
- Device emulation
- Device assignment
- Exception handling
- Instruction trapping
- Managing virtual exceptions
- Interrupt controller management
- Context switching
- Memory translation
- Managing multiple virtual address spaces
2-stage address translation
Traditional ARM Architecture

- ARM11 (ARMv6)
- Privilege Level 0
  - User mode
- Privilege Level 1
  - System
  - IRQ
  - FIQ
  - Supervisor
  - Abort

Virtualization Extensions (CPU, Memory, I/O) on ARM architecture are based on Security Extensions ("Trust Zone")
Overview of ARM Virtualization Extension

- Based on Security Extension
- CPU: new mode (HYP), and new Privilege Level (non-secure privilege level 2)
  - Support for sensitive instructions
  - Multiple interrupt vector tables
  - Hypervisor call
- Memory: Intermediate Physical Address (IPA)
  - Guest OS cannot access physical address space directly.
- I/O: Virtual Generic Interrupt Controller (vGIC)
  - Faster delivery of interrupts to VMs
Security Extensions ("TrustZone")

Non-Secure State
- Normal App
- Normal App
- Normal App

Non-Secure OS (ex: Linux)

Secure State
- Phone
- SMS
- Secure OS (ex: RTOS)
- Monitor

ARM Cortex-A8 and beyond
Privilege Levels in TrustZone (1)

Non-Secure State

- Privilege Level 0 of Non-Secure State
- Privilege Level 1 of Non-Secure State

Secure State

- Privilege Level 0 of Secure State
- Privilege Level 1 of Secure State
- Monitor mode

ARM Cortex-A8 and beyond
Privilege Levels in TrustZone (2)

Non-Secure State
- Privilege Level 0 of Non-Secure State
- Privilege Level 1 of Non-Secure State
- Privilege Level 2 of Non-Secure State

Secure State
- Privilege Level 0 of Secure State
- Privilege Level 1 of Secure State
- Monitor mode

ARM Cortex-A15 and beyond
Virtualization extensions to the ARMv7-A architecture:

- Available in Cortex A-15 / A-7 CPUs
- Hyp - New privilege level (for hypervisor)
  - GuestOS: SVC privilege level, Applications: USR privilege level
- 2-stage address translation (for OS and hypervisor levels)
- Complementary to TrustZone security extensions

Mechanisms to minimize hypervisor intervention for “routine” GuestOS tasks:

- Page table management
- Interrupt masking & Communication with the interrupt controller (GIC)
- Device drivers (hypervisor memory relocation)
- Emulation of Load/Store accesses and trapped instructions
  - Hypervisor Syndrome Register: Hype mode entry reason (syndrome)
- Traps into Hyp mode for ID register accesses & idling (WFI/WFE)
- System instructions to read/write key registers
ARMv7 CPU Virtualization

Non-Secure State

PL0
App App App App App App

PL1
Guest OS 1 Guest OS 2

PL2
Hypervisor

Secure State

RT App
RT App

RTOS
Monitor mode

ARM Cortex-A15

Virtualization in the ARM Architecture
ARM TrustZone (Secure System Partitioning)

- ARM TrustZone extensions introduce:
  - new processor mode: monitor
    - similar to VT-x root mode
    - banked registers (PC, LR)
    - can run unmodified guest OS binary in non-monitor kernel mode
  - new privileged instruction: SMI
    - enters monitor mode
  - new processor status: secure
  - partitioning of resources
    - memory and devices marked secure or insecure
    - in secure mode, processor has access to all resources
    - in insecure mode, processor has access to insecure resources only
  - monitor switches world (secure ↔ insecure)
  - really only supports one virtual machine (guest in insecure mode)
    - need another hypervisor and para-virtualization for multiple guests
NS: Not Secure - treated like an address line
ARMv7 processor modes

- **Non-Secure state**
  - EL0 User
  - EL1 Kernel
  - EL2 Hypervisor

- **Secure state**
  - EL0 User
  - EL1 Kernel

- **Monitor Mode (Secure EL3)**
Privilege levels

- Guest OS: same kernel/user privilege structure
- HYP mode: higher privilege than OS kernel level
  - hvc instruction (hypercall)
  - VMM controls wide range of OS accesses
- Hardware maintains TZ security (4th privilege level)

By default, HYP mode is disabled.
- Should be explicitly enabled in secure bootloader code.
Differences with Intel’s VT-x

- VT-x: Root-Mode is orthogonal to privilege levels (“rings”)
- ARM HYP: “just one more processor mode”
  - More privileged than existing “kernel” modes
  - Hypervisor must save guest VM’s register state
    - With VT-x, this is done automatically (by hardware)
Vector Tables

Non-Secure State

Privilege Level 0 of Non-Secure State

Privilege Level 1 of Non-Secure State

Hyp mode

Secure State

Privilege Level 0 of Secure State

Privilege Level 1 of Secure State

VT for Secure PLO&1

VT for Hyp mode

VT for Mon. mode

ARM Cortex-A15

Virtualization in the ARM Architecture
SVC vs. HVC (1)

- **SVC: Supervisor Call**
  - Software Interrupt (SWI) instruction: allow program to actively trigger an event to enter supervisor mode (from user mode)
  - Processor jumps to SVC vector stub (in kernel space)

- **CPU mode: Supervisor Mode**
  - svc_handler:
    - ...
    - ...
    - ...

- **CPU mode: User Mode**
  - svc #0x190
  - ...
  - pc

Virtualization in the ARM Architecture
SVC vs. HVC (2)

- **HVC: Hypervisor Call**
  - Instruction that allows program to actively trigger an event to enter hypervisor mode
  - Non-Secure PL1 → Non-Secure PL2
- Processor jumps to HVC vector stub (0x14)

### Vector Table with virtualization extension

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1C</td>
<td>FIQ</td>
</tr>
<tr>
<td>0x18</td>
<td>IRQ</td>
</tr>
<tr>
<td>0x14</td>
<td>Hypervisor Call</td>
</tr>
<tr>
<td>0x10</td>
<td>Data Abort</td>
</tr>
<tr>
<td>0x0C</td>
<td>Prefetch Abort</td>
</tr>
<tr>
<td>0x08</td>
<td>Supervisor Call</td>
</tr>
<tr>
<td>0x04</td>
<td>Undefined Instruction</td>
</tr>
<tr>
<td>0x00</td>
<td>Reset</td>
</tr>
</tbody>
</table>
Large Physical Address Extension

- Large Physical Address Extension
  - 64-bit descriptor entries, up to 3 levels
  - Input/Output addresses: up to 40 bits

- VMID: Virtual Machine Identifier
  - Identifies current VM, with its own Address Space Identifier (ASID – part of TLB maintenance tags)

- VTTBR: Virtual translation table base register
  - 8-bit VMID
Virtual Memory (1-stage translation)

- Without virtualisation, the OS owns the memory
  - Allocates areas of memory to the different applications
  - Virtual Memory commonly used in “rich” operating systems
Virtual Memory (2-stage translation)

Stage 1 translation owned by each Guest OS

Virtual address (VA) map of each App on each Guest OS

“Intermediate Physical” address map of each Guest OS (IPA)

Stage 2 translation owned by the VMM

Hardware has 2-stage memory translation

Tables from Guest OS translate VA to IPA

Second set of tables from VMM translate IPA to PA

Allows aborts to be routed to appropriate software layer

Physical Address (PA) Map
ARMv8 Privilege Model

- Each processor mode has its own linear address space, defined by a distinct page table.
- EL2 has its own translation regime → tag in TLB entries
  - No need to flush TLB in transitions bet. EL2 and other CPU modes.
ARM processor modes

The hypervisor enables the processor’s virtualization features in EL2 before switching to a VM. The VM will then execute normally, in EL0 and EL1

... until some condition is reached that requires the intervention of the hypervisor → trap into EL2

Each “interrupt” (IRQ, FIRQ) can be configured to trap directly into a VM’s EL1

Instead of going through EL2

E.g: system calls, page faults

Any state that needs to be saved & restored must be handled explicitly

Contrast: Intel VT-x → VM control block is automatically saved & restored, with a single instruction, when switching bet. Root – Non-Root modes.

EL2: simply a more privileged CPU mode
(i.e. can be used for purposes other than VM support)
Stage-1 and Stage-2 page table walk

VA (virtual address in VM) $\rightarrow$ gPA (guest physical address) $\rightarrow$ hPA (host physical address)
Steps for World Switch

1. Store all host GP registers onto EL2 stack
2. Configure vGIC and virtual timers for VM
3. Save host-specific CSRs onto EL2 stack
4. Load VM’s CSR’s (no impact on current execution, as EL2 uses its own CSRs – separate from host state)
5. Configure EL2 to trap FP operations for “lazy” context switching of VFP registers, trap interrupts, trap WFI/WFE (CPU halt) instructions, trap SMC instructions & specific CSR & debug-register accesses
6. Write VM-specific IDs into shadow ID registers
7. Set Stage-2 page table base register (VTTBR), enable Stage-2 address translation
8. Restore all Guest GP registers, and trap into either user or kernel mode for the VM.
### VM and Host State (Cortex-A15)

<table>
<thead>
<tr>
<th>Action</th>
<th>Nr.</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context Switch</td>
<td>38</td>
<td>General Purpose (GP) Registers</td>
</tr>
<tr>
<td></td>
<td>26</td>
<td>Control Registers</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>VGIC Control Registers</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>VGIC List Registers</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Arch. Timer Control Registers</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>64-bit VFP Registers</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>32-bit VFP Control Registers</td>
</tr>
<tr>
<td>Trap-and-Emulate</td>
<td>-</td>
<td>CP14 Trace Registers</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>WFI Instructions</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>SMC Instructions</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>ACTLR Access</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>Cache Ops. by Set/Way</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>L2CTLR / L2ECTLR Registers</td>
</tr>
</tbody>
</table>
Types of Hypervisors (1)

Hypervisor support in ARMv8 (aarch64):
- Dedicated exception level (EL2) for hypervisor
- Trapping exceptions that change core context/state
- Routing of exceptions and virtual interrupts
- 2-stage memory translation
- Dedicated exception (HVC) for Hypervisor Call
Types of Hypervisors (2)

- **Full virtualization**
  - Unmodified Guest OS
  - Guest I/O
    - emulated
    - ... or handled by virtualization-aware HW

- **Para-virtualization**
  - Guest OS is aware of the hypervisor
  - Privileged instructions are replaced by VMM hooks

- **Hybrid**
  - Use as much as possible hardware-assisted virtualization
  - Devices (network, block) para-virtualized or passthrough-ed
  - Use of emulation is very limited
Common hypervisors on ARM architecture
ARMv8 Virtualization Features

- 2nd stage of memory translation
  - Adds extra level of indirection between guest & physical memory
  - TLBs are tagged by Virtual Machine ID (VMID)

- Ability to trap access of most system registers
  - The hypervisor decides what it wants to trap

- Can handle IRQs, FIQs and asynchronous aborts
  - The guest doesn’t see physical interrupts firing

- Guests can call into EL2 mode (HVC instruction)
  - Allows para-virtualizated services

- Standard architecture peripherals are virtualization-aware
  - GIC and timer have specific features to help virtualization
EL2 in ARMv8

- EL2 is **not** a superset of NS-EL1
  - Orthogonal mode to EL1
  - Allows multiplexing of NS-EL1 guests on the hardware

- Own translation regime
  - Separate Stage-1 translation, no Stage-2 translation

- It would be difficult to run Linux in EL2
  - too many changes to be practical

- EL2 could be used as a ”world switch”
  - Between guests (bare-metal hypervisor/Type I)
  - Between host and guest (hosted hypervisor/Type II)
  - This makes the host a form of specialized guest ...
GIC: Generic Interrupt Controller

- Single interrupt controller in ARM architecture
- + Firmware: “Interrupt Distributor”
Virtualization of interrupts

- An interrupt might need to be routed to one of:
  - Current or different GuestOS
  - Hypervisor
  - OS/RTOS running in the secure TrustZone environment

- Physical interrupts are taken initially in the Hypervisor
  - If the Interrupt should go to a GuestOS:
    - Hypervisor maps a “virtual” interrupt for that GuestOS
Virtual Interrupt Controller

- ISR of GuestOS interacts with the virtual controller
  - Pending and Active interrupt lists for each GuestOS
  - Interact with the physical GIC in hardware
  - Creates Virtual Interrupts only when priority indicates it is necessary

- GuestOS ISRs therefore do not need calls for:
  - Determining interrupt to take [Read of Interrupt Acknowledge]
  - Marking the end of an interrupt [Sending EOI]
  - Changing CPU Interrupt Priority Mask [Current Priority]

- GIC has separate sets of internal registers:
  - Physical registers and virtual registers
    - Non-virtualized system and hypervisor access the physical registers
    - Virtual machines access the virtual registers
    - Guest OS functionality does not change when accessing the vGIC

- Hypervisor remaps virtual registers for use by GuestOS’es
  - Interrupts generate a hypervisor trap
Virtual interrupt sequence

- External IRQ (configured as virtual by the hypervisor) arrives at the GIC
- GIC Distributor signals a Physical IRQ to the CPU
- CPU takes HYP trap, and Hypervisor reads the interrupt status from the Physical CPU Interface
- Hypervisor makes an entry in register list in the GIC
- GIC Distributor signals a Virtual IRQ to the CPU
- CPU takes an IRQ exception, and Guest OS running on the virtual machine reads the interrupt status from the Virtual CPU Interface
Virtual I/O devices

- Memory-mapped devices
  - Read/write accesses to device registers have specific side-effects

- Virtual devices $\rightarrow$ emulation
  - Typically, read/write accesses have to trap to Hypervisor
    - Fetch & interpretation of emulated load/stores is performance-intensive
  - Syndrome: key information about an instruction
    - Source/destination register, Size of data transfer, ...
    - Available for some loads/stores (on abort)
      - If not available, then it is required to fetch the instruction for full emulation

- System MMU: 2$^{\text{nd}}$-stage address translation for devices
  - Allows devices to be programmed into Guest’s VA space
System MMU (IO-MMU)
Device emulation

- Platform devices are memory-mapped, and guest accesses to devices are subject to at least Stage 2 translation when virtualization is in effect.
- Guests can detect a platform device by reading its ID register, or by interrogating registers mentioned in the device tree.
  - The hypervisor can return dummy values for Guest reads and ignore writes, effectively giving the Guest the impression that the device does not exist on the platform.
  - When the device model has some data to deliver, the hypervisor raises a virtual IRQ (vIRQ).
    - The Guest OS responds by attempting to read hardware registers. The hypervisor traps these accesses and provides simulated responses.
Device assignment

- Need to hide from the Guest the fact that the device is at a different physical address
  - Alternatively, the hypervisor might choose to hide a device from a set of guests - either because the device is not present or has already been assigned to a different guest.
- Generate a different interrupt ID than that which the guest is expecting.
Type-I Hypervisor

- Guest Userspace
- Guest Userspace
- Guest Userspace
- Guest Kernel
- Guest Kernel
- Guest Kernel
- Hypervisor
Type-II Hypervisor

- Host Userspace
- Guest Userspace
- Guest Userspace
- Host Kernel
- Guest Kernel
- Guest Kernel
- HYP
- Switching Code

EL0
EL1
EL2

Virtualization in the ARM Architecture
Virtualization Host Extensions (VHE)

- Part of ARMv8.1 (AArch64 specific): expands the capability of EL2
  - Designed to improve the support of the Type-II hypervisors
  - Allows the host OS to be run at EL2
    - Significantly reduces the number of system registers shared between Host and Guest
  - The host OS requires minimal changes to run at EL2
    - Use HYP timer interrupt, instead of Guest’s timer interrupt
    - User-space still runs at EL0
    - Host has no software running at EL1
- VHE provides a mechanism to access the extra EL2 register state transparently.
  - Simply providing extra EL2 registers is not sufficient to run unmodified OSes in EL2, because existing OSes are written to access EL1 registers.
Type-II Hypervisor with VHE
VHE: Impact on Hypervisor (kvm)

- Reduced save/restore of host system registers
  - Only 4 registers
- “Lazy” save/restore of Guest system registers
  - Deferred until actually needed by Host, or upon VM switch
- Reduced interrupt latency
  - Less state to be restored before handling the interrupt
- No trap to enter hypervisor (normal function call!)
- No HYP mappings needed
  - The kernel runs at EL2
Nested Virtualization

- Part of ARMv8.3
- Allows an hypervisor in a VM
  - Unmodified guest hypervisor running in NS EL1
  - VM thinks it runs at “virtual” EL2 (using VHE)
    - VM uses EL1 register accesses to access EL2 registers (HCR_EL2.NV, Shadow EL1 registers)
    - Very few traps needed (EL1 → EL2, “eret”)
- Implementation of a host hypervisor required
  - Running at EL2

Use cases:
- Develop/test/debug hypervisor in VM
- IaaS hosting private cloud
KVM: Kernel-based Virtual Machine

- Hosted (Type-II) hypervisor
  - Unlike Xen: Type-I
- Use of assisted hardware virtualization
- Devices are
  - emulated (QEMU)
  - para-virtualized (VIRTIO)
- All CPUs are using the same scheduler
  - guest vCPU is a task for the host OS
- Resource management can be done using cgroups
  - Standard way in Linux to control resources
KVM Architecture

KVM guest
- Applications
  - File system and block devices
  - Drivers
- vcpu0 ... vcpu N
- iotread

KVM (kvm.ko)
- File system and block devices
- Physical drivers

Hardware
- cpu0 ... cpu N
- Disk

Linux kernel

KVM guest's kernel

Hardware emulation (QEMU)
- Only one thread can run QEMU code at any time (qemu_mutex)
- Generates I/O requests to the host on guest's behalf and handles events

Virtualization in the ARM Architecture
Virtualization in the ARM Architecture

KVM architecture with ARMv8.0-A

State to save/restore:
- Stage-2 translation table
- Trap configuration
- General-purpose registers
- System control registers
- FP registers
- GIC configuration
- Timer configuration
KVM architecture with ARMv8.1-A

ARMv8.1 features:
- 16-bit VMIDs
- VHE
  (expanded EL2)
Xen: Para-virtualization

- Dom0
  - PV backends
  - HW drivers

- DomU
  - PV Frontends

- Xen

- Hardware
Xen: Driver Domains

Dom0

Disk Driver Domain

Network Driver Domain

DomU

Toolstack

BlockBack

Disk Driver

NetBack

Network Driver

NetFront

BlockFront

Xen

Hardware

Virtualization in the ARM Architecture
Sources (1)

- David Brash, *Extensions to the ARMv7-A Architecture*, HotChips 2010
- Roberto Mijat and Andy Nightingale, *Virtualization is Coming to a Platform Near You: The ARM Architecture Virtualization Extensions and the importance of System MMU for virtualized solutions and beyond*, ARM White paper, 2011
Sources (2)

- AArch64 virtualization Documentation

- AArch64 virtualization: Hypervisor software
  - https://developer.arm.com/docs/100942/latest/hypervisor-software

- Julien Grall: “Hypervisors on ARM: Overview and Design choices”, Root Linux Conference 2017
  - Slides: https://www.slideshare.net/xen_com_mgr/rootlinux17-hypervisors-on-arm-overview-and-design-choices-by-julien-grall-arm
  - Video: https://www.youtube.com/watch?v=jZNXtqFJpuc

- ARMv8.1-A: https://goo.gl/Ox4thV
- ARMv8.2-A: https://goo.gl/0Ns37U
- ARMv8.3-A: https://goo.gl/CJv1n0