虚擬化技術
Virtualization Technique

Hardware Support
Virtualization
ARM's Virtualization Extension
• Review on system virtualization
  ▪ Definition of Virtual Machine Monitor
  ▪ Problems when we try to implement VMM
  ▪ Software solution
  ▪ Hardware solution

• Overview of ARM architecture
  ▪ Brief introduction of ARM architecture
  ▪ Security Extension
  ▪ Virtualization Extension
• Support for CPU virtualization
  ▪ Support for Sensitive Instruction
    • Native-run
    • Emulation support
  ▪ Support for multiple Vector Table
    • Vector Table Base Address Registers
  ▪ Hypervisor Call
    • New entry(0x14) on vector table
Agenda

• Support for Memory virtualization
  ▪ Intermediate Physical Address
  ▪ VMID

• Support for I/O virtualization
  ▪ Virtual GIC

• Case study: Citrix Xen for ARM Cortex-A15
REVIEW ON SYSTEM VIRTUALIZATION
**Definition of VMM**

- A virtual machine monitor can be constructed if the set of sensitive instructions is a subset of the set of privileged instructions.
- **Proof shows**
  1. **Equivalence**
     - by interpreting privileged instructions and executing remaining instructions natively
  2. **Resource control**
     - by having all instructions that change resources trap to the VMM
  3. **Efficiency**
     - by executing all non-privileged instructions directly on hardware
- A key aspect of the theorem is that it is easy to check
Implement models of VMM

Type-I

Type-II
Problems in implementation

- CPU virtualization
  - Non-virtualizable CPU
  - Vector table issue
- Memory virtualization
  - Translation from GVA to HPA
- I/O virtualization
  - Emulate various I/O devices
Software Solution

• CPU virtualization
  ▪ Non-virtualizable CPU
    • → Patch GOS / Dynamic Binary Translation
  ▪ Vector table issue
    • → Modify host’s VT to deliver interrupts to host or guests

• Memory virtualization
  ▪ Translation from GVA to HPA
    • → Implement by software and save the translated result in Shadow Page Table

• I/O virtualization
  ▪ Emulate various I/O devices
    • → Emulated by Software (like QEMU)
Disadvantage of software solution

- CPU virtualization
  - Patch GOS
    - You need source code. It’s non-portable.
  - Dynamic Binary Translation
    - Slow down the performance on runtime
  - Vector table issue
    - Difficult to route interrupts to hypervisor and many GOSs

- Memory virtualization
  - Translation from GVA to HPA
    - Waste time when generating to translation result

- I/O virtualization
  - Emulate various I/O devices
    - Quite slow when emulating I/O devices
    - I/O devices are too diversity to cover all of them
Hardware solution

• CPU virtualization
  ▪ Non-virtualizable CPU
    • → Let CPU become virtualizable
  ▪ Vector table issue
    • → With different registers for hypervisor and guest.

• Memory virtualization
  ▪ Translation from GVA to HPA
    • → Translate VA-IPA-PA by MMU rather than software

• I/O virtualization
  ▪ Emulate various I/O devices
    • → Directly I/O access
OVERVIEW OF ARM ARCHITECTURE
Brief introduction of ARM architecture
ARM architecture

• ARM architecture is a well-known CPU architecture in embedded system, mobile devices, and low-power consumption devices.

• Unlike other CPU vendors, ARM doesn’t produce and ship their own chip. ARM license their IP to other SoC design house and let them to combine ARM CPU Core with their other chips.

• ARM also allow SoC design house to change CPU’s ISA in their chips.
Roadmap of ARM’s CPUs

• ARMv4
  ▪ ARM7: Without MMU

• ARMv5
  ▪ ARM9: With MMU

• ARMv6
  ▪ ARM11: With MPCore support
Roadmap of ARM’s CPUs

- **ARMv7**
  - With security extension, but without virtualization extension
    - Cortex-A8
    - Cortex-A9 / Cortex-A5
  - With security and virtualization extension, with LPPE
    - Cortex-A15 / Cortex-A7 (with big-LITTLE support)

- **ARMv8**
  - 64-bit support
    - Cortex-A57 / Cortex-A53 (aka. Cortex-A50 series)
Traditional ARM architecture

- Privilege Level 0
- Privilege Level 1
- ARM 11 (ARMv6)
• Privilege Level 0:
  ▪ User mode
• Privilege Level 1:
  ▪ System mode
  ▪ IRQ mode
  ▪ FIQ mode
  ▪ Undefined mode
  ▪ Supervisor mode
  ▪ Abort mode
Overview : Security Extension
Security Extension

- Security Extension, a.k.a. Trust-Zone, is a system-wide approach to security on high performance computing platforms for a huge array of applications including secure payment, digital rights management (DRM), enterprise and web-based services.
- Security extension can make trusted application running on secure state and non-trusted application running on non-secure state.
- When you are in the Non-Secure State, you cannot access the memory which is allocated for Secure State.
- It has been introduced since from ARM Cortex-A8
Security Extension

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 Level of Security Extension

- Secured apps and secured OS runs on secure state
  - Monitor runs on monitor mode which is on Privilege Level 1 of Secure State (Secure State PL1).
  - Secured OS runs on Privilege Level 1 of Secure State (Secure State PL1).
  - Secured apps run on Privilege Level 0 of Secure State (Secure State PL0).
Privilege Level of Security Extension

- Normal apps and normal OS runs on non-secure state
  - Normal apps (e.g. Angry-Bird, Browser, …) run on **Privilege Level 0 of Non-Secure State** (Non-Secure PL0).
  - Normal OS (e.g. Linux) runs on **Privilege Level 1 of Non-Secure State** (Non-Secure PL1).
Privilege Level of Security Extension

• [Note] Privilege Level in ARM architecture
  ▪ In ARM architecture, when the number of privilege level is higher means that it has more privilege to access hardware resource.
Privilege Level of Security Extension

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
Overview : Virtualization Extension
Virtualization Extension

• Virtualization extension is the hardware support for virtualization on ARM architecture.
• From ARM Cortex-A15 and beyond, ARM architectures are including virtualization extension.
• We can consider that **virtualization extension extends security extension for virtualization**.
• Virtualization extension includes three major parts:
  ▪ CPU virtualization extension
  ▪ Memory virtualization extension
  ▪ I/O virtualization extension
In CPU virtualization extension, ARM adds a new mode and a new Privilege Level. It calls “Hyp mode” which is running on Non-Secure Privilege Level 2.

In Memory virtualization extension, ARM adds “Intermediate Physical Address”, let Guest OS cannot access physical address directly.

In I/O virtualization extension, ARM adds “Virtual Generic Interrupt Controller” interface to deliver interrupt in more faster way.
Privilege Level of Virtualization Extension

**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
Support for Sensitive Instruction
Support for multiple Vector table
Hypervisor call

SUPPORT FOR CPU VIRTUALIZATION
CPU virtualization extension

• It lets Guest OS run on Non-Secure Privilege Level 1 and Non-Secure Privilege Level 0.
  ▪ Kernel of Guest OS runs on Non-Secure PL1.
  ▪ Userspace of Guest OS runs on Non-Secure PL0.

• It allows that most of (not all of them) sensitive instructions can native-run on Non-Secure PL1 without trap-and-emulation.

• There are still some of sensitive instruction which need “trap-and-emulation”. When these instructions execute on Non-Secure PL1, it will be trapped into Hyp mode (in Non-Secure PL2).
Privilege Level of Virtualization Extension

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
CPU virtualization

Non-Secure State:
- App
- App
- App
- App
- App
- App

Secure State:
- RT App
- RT App

PL0:
- Guest OS 1
- Guest OS 2

PL1:
- Hypervisor

PL2:
- RTOS
  - Monitor mode

ARM Cortex-A15
• There is only one mode in Non-Secure Privilege Level 2.
  ▪ That is Hyp mode
  ▪ Hyp mode has its own SP, SPSR, ELR register
  ▪ Most of critical feature of hardware-assistant CPU virtualization is executed in Hyp mode.
Hyp mode

![Diagram showing modes, privilege levels, and security states]

Figure B1-1 Modes, privilege levels, and security states

• Hyp mode is off in default. If Hyp mode is off, all virtualization extension will be set off.

• If you want to open Hyp mode, you have to use secure software (ex: Bootloader) running on secure state turn the Hyp mode on.
Hyp mode is different with x86’s VT-x

• In VT-x of x86 architecture, “root-mode” is orthogonal to x86 rings.
• However, in ARM’s virtualization extension, Hyp mode is one mode of all modes in ARM architecture.
• ARM’s hyp mode is strictly more privileged than the existing kernel modes. ARM requires the hypervisor to save guest register state, while on x86 this is done automatically by hardware.

Ref: Prashant Varanasi, Gernot Heiser, “Hardware-Supported Virtualization on ARM”, APSys 2011
Instruction emulation

• There are still some instructions which cannot execute natively.
  ▪ Guest OS’s Load/Store
  ▪ The instruction’s behavior will effect other Guest OS

• In virtualization extension, they will be trapped into hypervisor automatically when you execute these instructions on Non-Secure Privilege Level 1
  ▪ There is **NO** necessary to patch guest OS for adding hypervisor before executing a critical instructions because it will be trapped by hardware automatically.
Instruction emulation

- When some instruction is trapped to Hyp mode, the hardware will also preserve some information on “Hypervisor Syndrome Register” (HSR) which is also a part of virtualization extension.
- With the help of HSR, the hypervisor can know the entry reason and emulate it.
  - In the past, hypervisor still need these information. In order to get these information, hypervisor designer have to some software way to get these information.
  - With the assistance from hardware, hypervisor can get these information more easier and more faster.
Support for Sensitive Instruction
Support for multiple Vector table
Hypervisor call

SUPPORT FOR CPU VIRTUALIZATION
Vector table for Monitor mode

- In **Security Extension**, ARM provides two vector tables. One of them is the original one. Another one is provided for the monitor mode.
  - The base address of monitor mode’s vector table is saved in MVBAR (Monitor Vector Base Address Register)
  - Only accessible in Monitor mode
  - When booting, software have to set Monitor mode’s vector table in Monitor mode.
Vector table for Hyp mode

• In Virtualization Extension, ARM provides different vector table for Hypervisor.
  ▪ The base address of Hyp mode’s vector table is saved in HVBAR(Hypervisor Vector Base Address Register)
  ▪ Only accessible in Monitor mode and Hyp mode
  ▪ When booting, software have to set Hyp mode’s vector table in Monitor mode.
In **Virtualization Extension**, ARM provides a register to save the vector table base address of Guest OS (which is running on Non-Secure PL0/1).

- The base address of Non-Secure PL0/1’s vector table is saved in VBAR (Vector Base Address Register).
- VBAR saves the base address of Non-Secure PL0/1’s vector table.
Overview of Vector Tables of all modes

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 PL0&1
- VT for Mon. mode

ARM Cortex-A15
Support for Sensitive Instruction
Support for multiple Vector table
Hypervisor call

SUPPORT FOR CPU VIRTUALIZATION
Before we introduce Hypervisor Call (HVC), we should recall some knowledge about Supervisor Call (SVC) in normal OS.

In previous microarchitecture version of ARM, it is called Software Interrupt (SWI).

Supervisor Call is a instruction which can trigger a software interrupt. If the program execute “svc” in the user space (Privilege Level 0), the processor will move into the vector stub of SVC which is in the kernel space.

SVC is a way to actively trigger event to enter the supervisor mode from user mode.
How does SVC work?
What is HVC?

• HVC, a.k.a. Hypervisor Call, is a instruction which will actively trigger event to enter Hyp mode(Non-Secure PL2) from Guest OS(Non-Secure PL1).

• The relationship between Guest OS and Hypervisor in HVC is the same as the relationship between User-space and Kernel-space in SVC.

• In vector table, the interrupt from HVC will be taken into the 0x14 vector stub.

• With the help of HVC, hypervisor designer can be easily to implement “Hypercall”
## Vector stub of HVC

### Vector Table without virtualization extension

<table>
<thead>
<tr>
<th>Vector</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Reset</td>
</tr>
<tr>
<td>0x04</td>
<td>Undefined Instruction</td>
</tr>
<tr>
<td>0x08</td>
<td>Supervisor Call</td>
</tr>
<tr>
<td>0x0C</td>
<td>Prefetch Abort</td>
</tr>
<tr>
<td>0x10</td>
<td>Data Abort</td>
</tr>
<tr>
<td>0x14</td>
<td>(Reserved, Not used)</td>
</tr>
<tr>
<td>0x18</td>
<td>IRQ</td>
</tr>
<tr>
<td>0x1C</td>
<td>FIQ</td>
</tr>
</tbody>
</table>

### Vector Table with virtualization extension

<table>
<thead>
<tr>
<th>Vector</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Reset</td>
</tr>
<tr>
<td>0x04</td>
<td>Undefined Instruction</td>
</tr>
<tr>
<td>0x08</td>
<td>Supervisor Call</td>
</tr>
<tr>
<td>0x0C</td>
<td>Prefetch Abort</td>
</tr>
<tr>
<td>0x10</td>
<td>Data Abort</td>
</tr>
<tr>
<td>0x14</td>
<td>Hypervisor Call</td>
</tr>
<tr>
<td>0x18</td>
<td>IRQ</td>
</tr>
<tr>
<td>0x1C</td>
<td>FIQ</td>
</tr>
</tbody>
</table>
How does HVC work?
Memory

Hypervisor

Guest OS

CPU mode: NS PL2

CPU PL: NS PL1

hvc_handler:
...
...

b hvc_handler

Vector Table

PC

hvc #0x190
...

...
Intermediate Physical Address
VMID
SUPPORT FOR MEMORY VIRTUALIZATION
Translation Regime

• Recall that
  - Host OS & guest OS’s execute in non-secure PL1&0.
  - Hypervisor runs in non-secure PL2
  - Other RTOS’s run in secure mode.

• Different software reference different page tables, and there are corresponding MMU to manage the translation process.
  - This design can separate different translation request handling process. Requests from guest OS must be intercepted by VMM, and requests from hypervisor itself could be handled in the original way.
Memory Translation on ARMv7

Figure B3-2 Memory translation summary, with Virtualization Extensions
What is IPA?

- **Intermediate physical address**
  - IPA is the output of the stage 1 translation, and is also the input of the stage 2 translation.

- **Stage 1 maps the Virtual Address (VA) to an Intermediate Physical Address (IPA).**
  - Typically, the Guest OS configures and controls this stage, and believes that the IPA is the Physical Address (PA).

- **Stage 2 maps the IPA to the Physical Address (PA).**
  - Typically, the hypervisor controls this stage, and a Guest OS is completely unaware of this translation.
  - VMM take whole control at this stage.
How VA-IPA-PA works?

- If the translation regime only provides one stage translation regime, then IPA is equal to physical address.
- Stage 1 translation must be in 64-bits format.
- LPAE (Large Physical Address Extension)
How VA-IPA-PA works?

- **Stage 1**
  - The input address range is 32 bits
  - The output address range is 40 bits. (1TB)
  - With virtualization extension, the output is IPA; otherwise, it’s PA.

![Diagram of Stage 1 address translation using Long-descriptor format](image)

*Figure B3-12 General view of stage 1 address translation using Long-descriptor format*
How VA-IPA-PA works?

- **Stage 2**
  - Supported only if virtualization extension is included.
  - The input address range must be 40 bits.
  - The output address range is up to 40 bits.

Figure B3-13 General view of stage 2 address translation, Long-descriptor translation table format
Large Physical Address Extension

- 64-bit descriptor entries
- Up to three levels of address lookup.
- Input addresses of up to 40 bits, when used for stage 2 translations.
- Output addresses of up to 40 bits.
- 4KB assignment granularity across the entire PA range.
- No support for domains, all memory regions are treated as in a Client domain.
- 64-bit table entries.
- Fixed 4KB table size, unless truncated by the size of the input address space.
Intermediate Physical Address
VMID

SUPPORT FOR MEMORY VIRTUALIZATION
What is VMID?

- **Virtual machine identifier**
  - identifies the current virtual machine, with its own independent ASID space.
  - ASID (address space identifier): part of TLB maintenance.
    - With ASID, TLB could identify entries belong to different process

- **Only used in LPAE.**

- **When to use VMID**
  - The TLB entries include this VMID information, meaning TLBs do not require explicit invalidation when changing from one virtual machine to another, if the virtual machines have different VMIDs.
  - For stage 2 translations, all translations are associated with the current VMID, and there is no concept of global entries.
How VMID works?

- When a translation walk is requested, the VMID will be checked to make sure the PTB, which VTTBR points to, is correct.

- **VTTBR**
  - Virtual translation table base register
  - 64 bits with 8-bit VMID
Virtual GIC

SUPPORT FOR I/O VIRTUALIZATION
GIC, a.k.a. Generic Interrupt Controller, is the only one interrupt controller in ARM architecture.

There is a firmware, called “Interrupt Distributor”.

In Interrupt Distributor, it save the information that which kinds of interrupt should be routed into which kind of state.

Interrupt distributor should be setting in booting time. Then, we don’t need to take care about GIC.

In next slide, we will show you that how it works.
• In software solution of system virtualization, I/O device emulation is always heavyweight.
• In order to avoid emulation of I/O device, ARM provide virtual interrupt with the help of new hardware component, the virtual CPU interface.
• Using these hardware support, it can let guest to acknowledge and clear interrupts without trapping into the hypervisor.

Ref: Prashant Varanasi, Gernot Heiser, “Hardware-Supported Virtualization on ARM”, APSys 2011
Virtual Interrupt Distributor

- The hypervisor must still emulate the interrupt distributor; all guest accesses to it trap.
- This is not expected to cause performance issues, as the distributor is normally only accessed at boot time to register drivers for particular interrupts and route them to specific (virtual) CPUs.

Ref: Prashant Varanasi, Gernot Heiser, “Hardware-Supported Virtualization on ARM”, APSys 2011
Virtual Interrupt Distributor

- Virtual Interrupt Distributor’s job is the category the interrupt group. Choose which kinds of interrupt should be routed to hypervisor’s vector table and which kinds of interrupt should be routed to Guest OS’s vector table.
- ARM provides “Virtual CPU interface” in GIC to help hypervisor designer to implement Virtual Interrupt Distributor.
Virtual Interrupt Distributor

• Once hypervisor’s “Virtual Interrupt Distributor” has set the interrupt controller in booting time. When interrupt occurs in running time, the interrupt which is categorized as “Direct route to GOS” will directly be routed to Guest OS’s vector table and handle by Guest OS without the help of hypervisor.

• Only the interrupt which is categorized as “Need to route to hypervisor” will be routed to hypervisor’s vector table and handle by hypervisor.
Booting time
Distributor

Hypervisor

Virtual Distributor

CPU Interface

Virtual CPU Interface

GOS

GOS Secure

Interrupt

Running time

GIC

CPU

Hypervisor

Running time

Secure
ARM will support I/O virtualization via a system MMU (SMMU) similar to the IOMMU on x86, but this is a platform feature, and ARM has to date only released their extensions to the core ISA.

Ref: Prashant Varanasi, Gernot Heiser, “Hardware-Supported Virtualization on ARM”, APSys 2011
Citrix Xen for ARM Cortex-A15

CASE STUDY
Review: Xen on x86

• Xen is a Type-I hypervisor. Xen directly runs above hardware. And the guests of Xen are run above Xen.

• The guest OSs run above Xen have different privilege level.
  ▪ Dom0: Have higher privilege.
  ▪ DomU: Other guest OS. Have lower privilege

• Xen support full-virtualized guest OS and para-virtualized guest OS.
  ▪ Full virtualized GOS is called as HVM
  ▪ Para virtualized GOS is called as PV
Review: Xen’s architecture (PV)
Review: Xen’s architecture (HVM)
Design goal of ARM version

- PV one GOS for all physical machine
- Use PV interface for I/O
- Rearrhitected for the modern age
  - No more QEMU
  - No Shadow Page Table
• Run most of instructions natively
• Only emulate some instructions in hypervisor
• Same entry point on native and on Xen
• Use HVC to implement hypervisor call in PV guest
Memory virtualization

- Use two layer translation (VA-IPA-PA)
- Remove SPT related codes:
  - 10,721 lines of code have been removed
- Use VMID to separate different VM’s memory
- In HVM guest only
  - No PV MMU calls
I/O virtualization

• Use virtual GIC to support I/O virtualization
• In PV Guest
  ▪ No emulated device
  ▪ Use Para-virtualization interface for I/O
• In HVM Guest
  ▪ Use device tree to discover Xen presence
  ▪ Remove useless device in Xen’s device tree
  ▪ Simple device emulation can be done in Xen
Device tree of Xen

hypervisor {
    compatible = "xen,xen", "xen,xen-4.2";
    reg = <0xb0000000 0x20000>;
    interrupts = <1 15 0xf08>;
};
References

• Paper resources:
  ▪ Prashant Varanasi, Gernot Heiser, “Hardware-Supported Virtualization on ARM”, APSys 2011

• Web resources:
  ▪ Linaro connect: Introduction to Xen on ARM http://www.slideshare.net/xen_com_mgr/linaro-connect-xen-on-arm-update

• Architecture manual resources: