Embedded Virtualization applied in Mobile Devices

Jim Huang (黃敬群) <**jserv**@0xlab.org>

Developer, 0xlab – `http://0xlab.org/`

June 20, 2012 / Acadamia Sinica

Rights to copy

Goals of This Presentation

- Give you an overview about
  - device virtualization on ARM
  - Benefit and real products
  - Android specific virtualization consideration
  - doing virtualization in several approaches
- We will not discuss
  - language runtimes
  - In-place multi-environment runtime

Agenda

(1) Motivations

(2) ARM Virtualization

(3) Embedded Virtualization Implementations

(4) Guest OS specific issues

Motivations to enable virtualization for embedded devices

Definition

"**virtualization** is "a technique for hiding the physical characteristics of computing resources from the way in which other systems, applications, or end users interact with those resources. "

– Wikipedia

Server Virtualization::Benefits

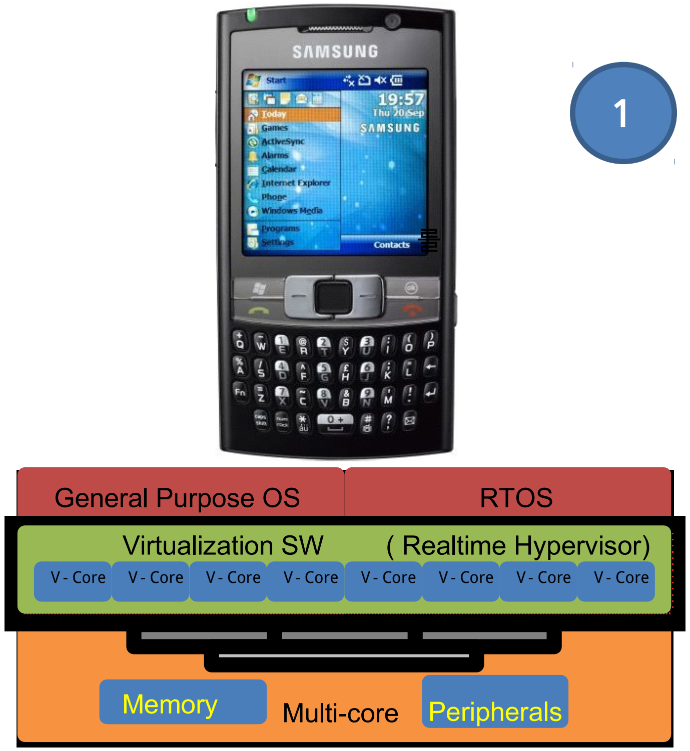
- Workload consolidation
  - Increase server utilization
  - Reduce capital, hardware, power, space, heat costs
- Legacy OS support
  - Especially with large 3rd-party software products
- Instant provisioning
  - Easily create new virtual machines
  - Easily reallocate resources (memory, processor, IO) between running virtual machines
- Migration
  - Predicted hardware downtime
  - Workload balancing

Embedded Virtualization::Benefits

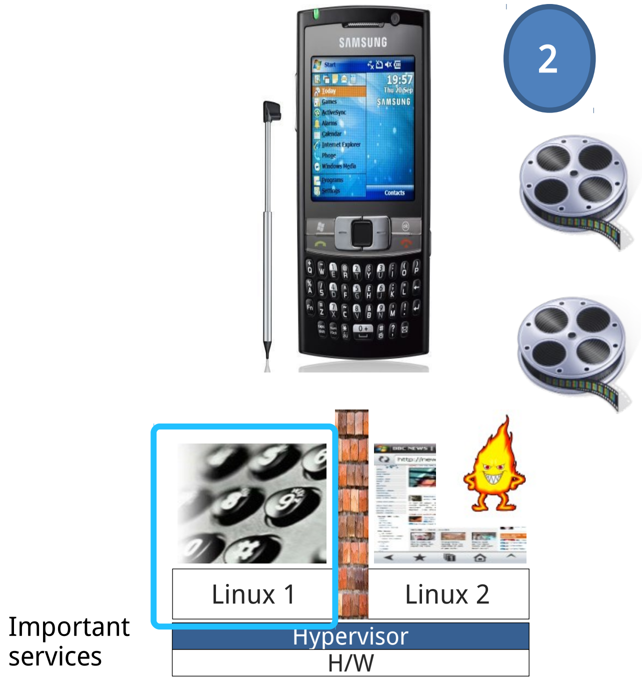
- Workload consolidation
- Flexible resource provisioning
- License barrier
- Legacy software support
  - Especially important with dozens or hundreds of embedded operating systems, commercial and even home-brew
- Reliability
- Security

Why?

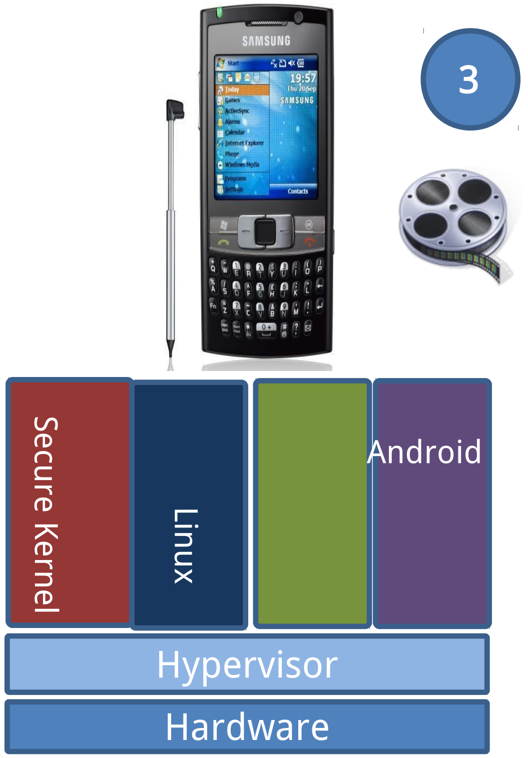
- (1) **Hardware Consolidation**
  - *Application Processor* and *Baseband Processor* can share multicore ARM CPU SoC to run both Linux and RTOS efficiently.
- (2) **OS Isolation**
  - important call services can be effectively separated from downloaded third party applications by virtualized ARM combined with access control.



AP SoC + BP SoC → Consolidated Multicore SoC



Secure Smartphone



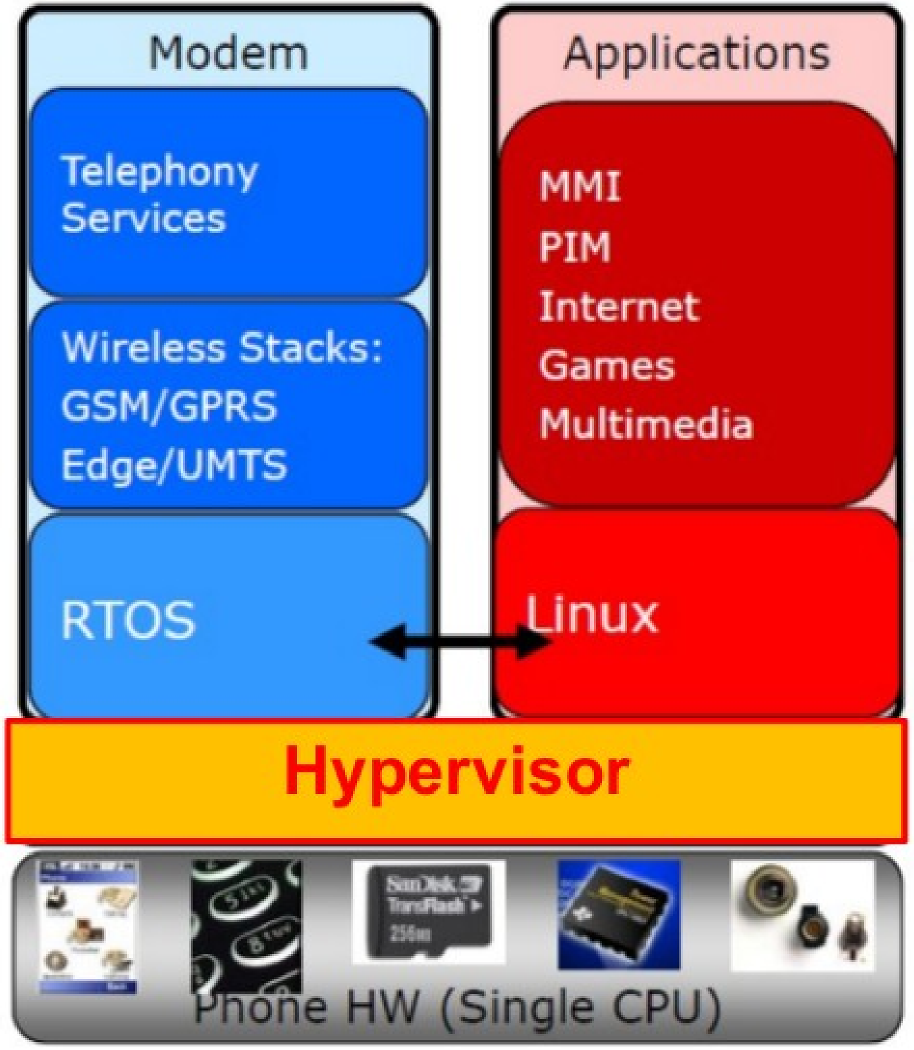
Rich Applications from Multiple OS

- (3) **Rich User Experience**
  - multiple OS domains can run concurrently on a single smartphone.

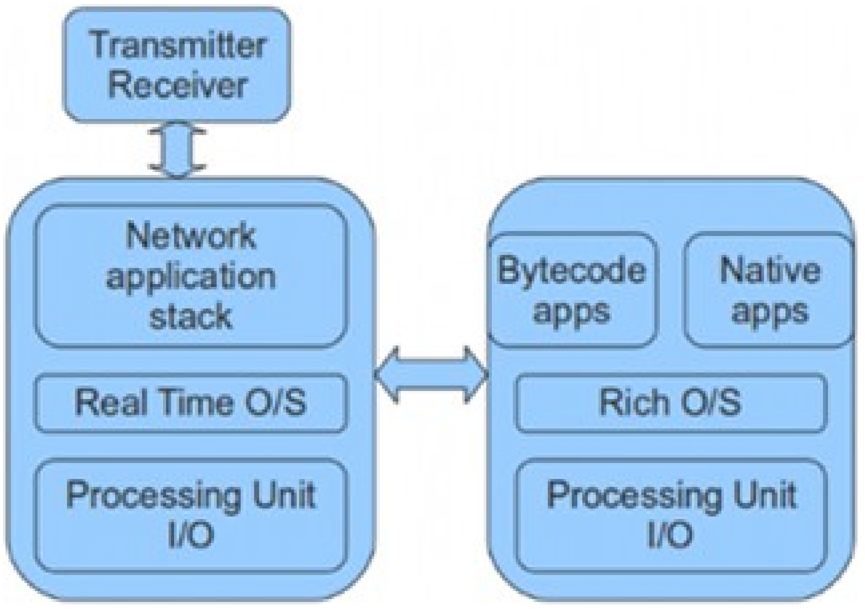
Source: **Xen ARM Virtualization**, Xen Summit Asia 2011 by Dr. Sang-bum Suh, Samsung

Use Case: Low-cost 3G Handset

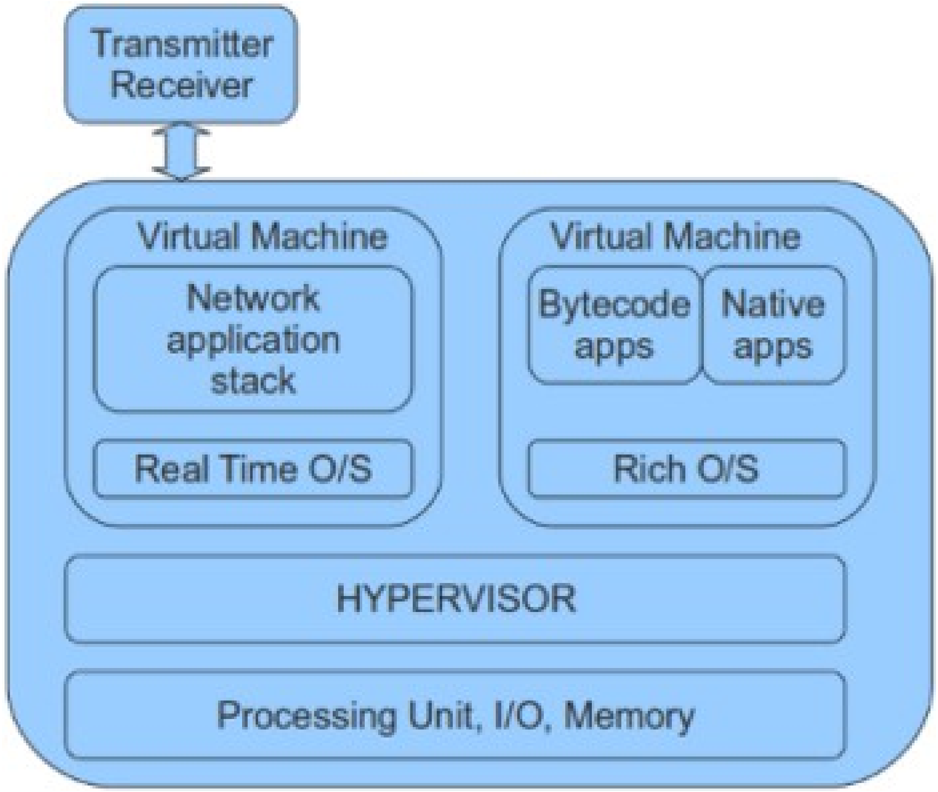
- Mobile Handsets
  - Major applications runs on Linux
  - 3G Modem software stack runs on RTOS domain
- Virtualization in multimedia Devices
  - Reduces BOM (bill of materials)
  - Enables the Reusability of legacy code/applications
  - Reduces the system development time
- Instrumentation, Automation
  - Run RTOS for Measurement and analysis
  - Run a GPOS for Graphical Interface
- Real cases: Motorola Evoke QA4



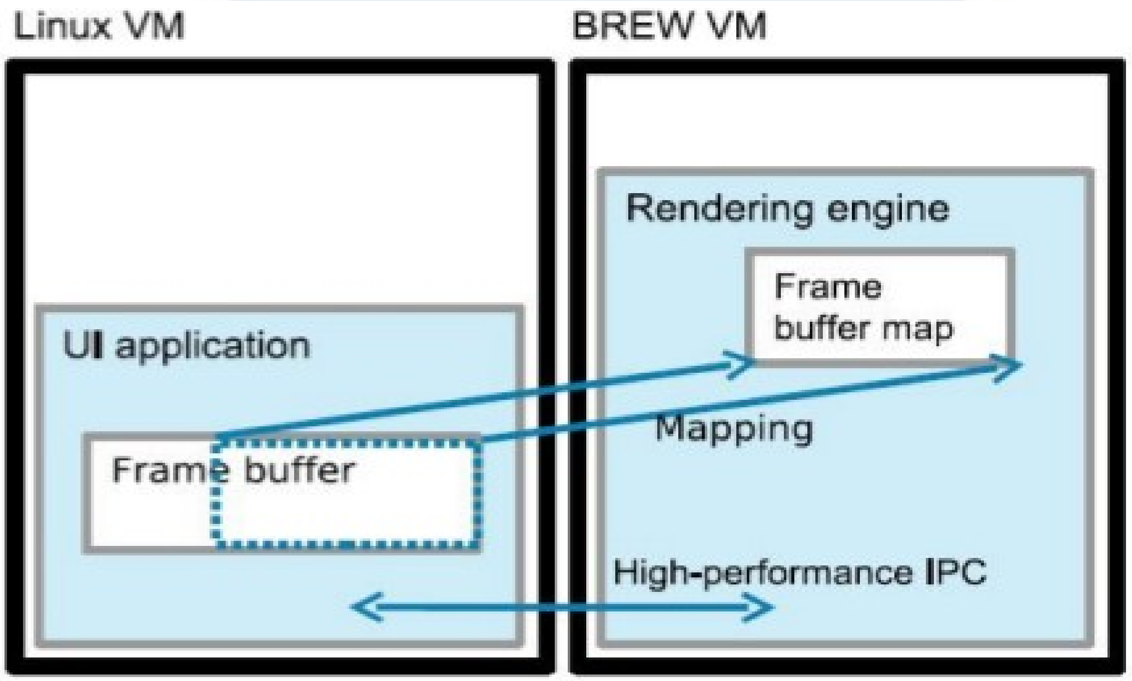
original mobile phone:
two CPUs required



with Virtualization: single chip



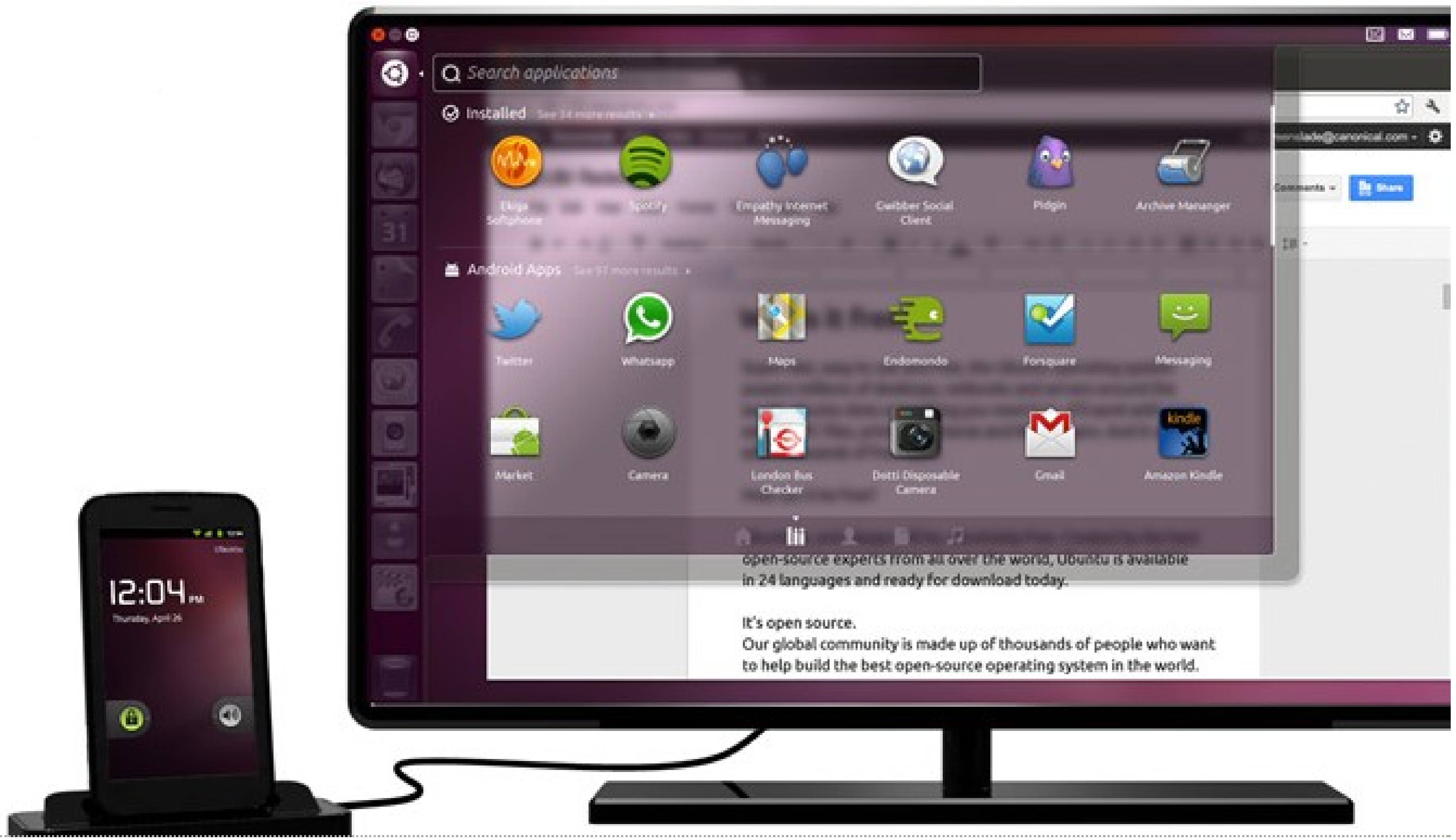
- Evoke’s UI functionalities including the touch screen is owned by the Linux apps while video rendering uses a rendering engine running on BREW.
- When a user requests a BREW app, Linux communciates with BREW in the other VM to start up the app. The BREW obtains access to the screen by using a frame buffer from a shared-memory mapping.



Example: **Ubuntu for Android**

Mobile (Android) and Desktop (Ubuntu) Virtualization in One Device by VMware and Canonical

http://www.ubuntu.com/devices/android



open-source experts from all over the world, Ubuntu is available in 24 languages and ready for download today.

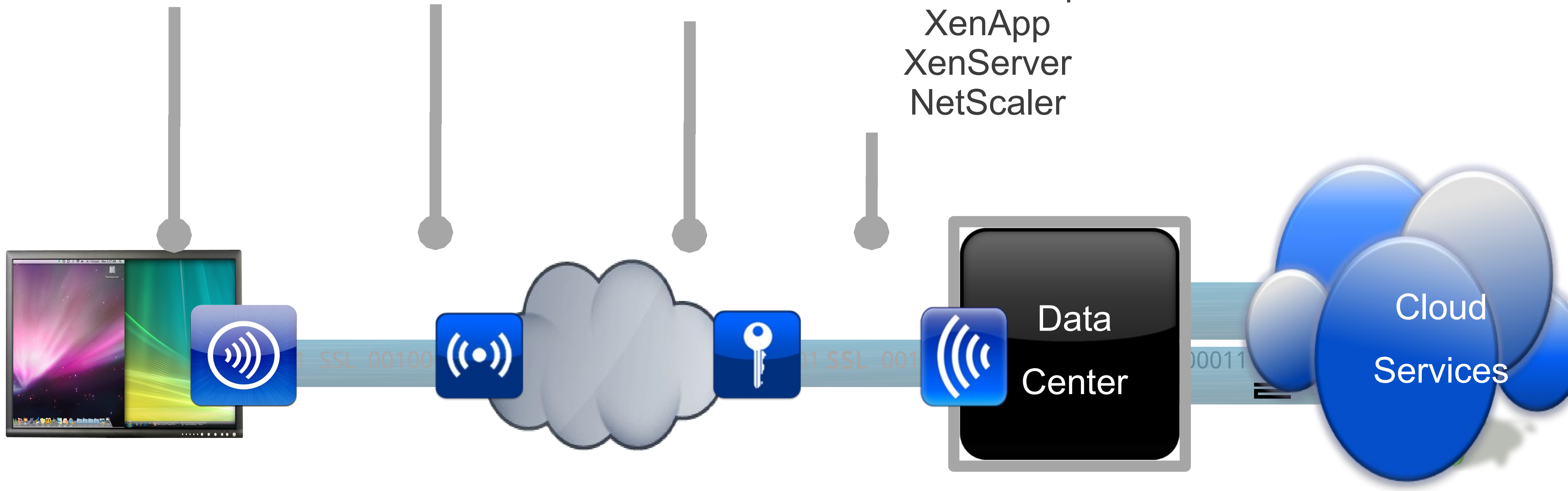
It's open source.
Our global community is made up of thousands of people who want to help build the best open-source operating system in the world.

Use Case: **Nirvana:**

The Convergence of Mobile and Desktop Virtualization in One Device by OKLabs + Citrix

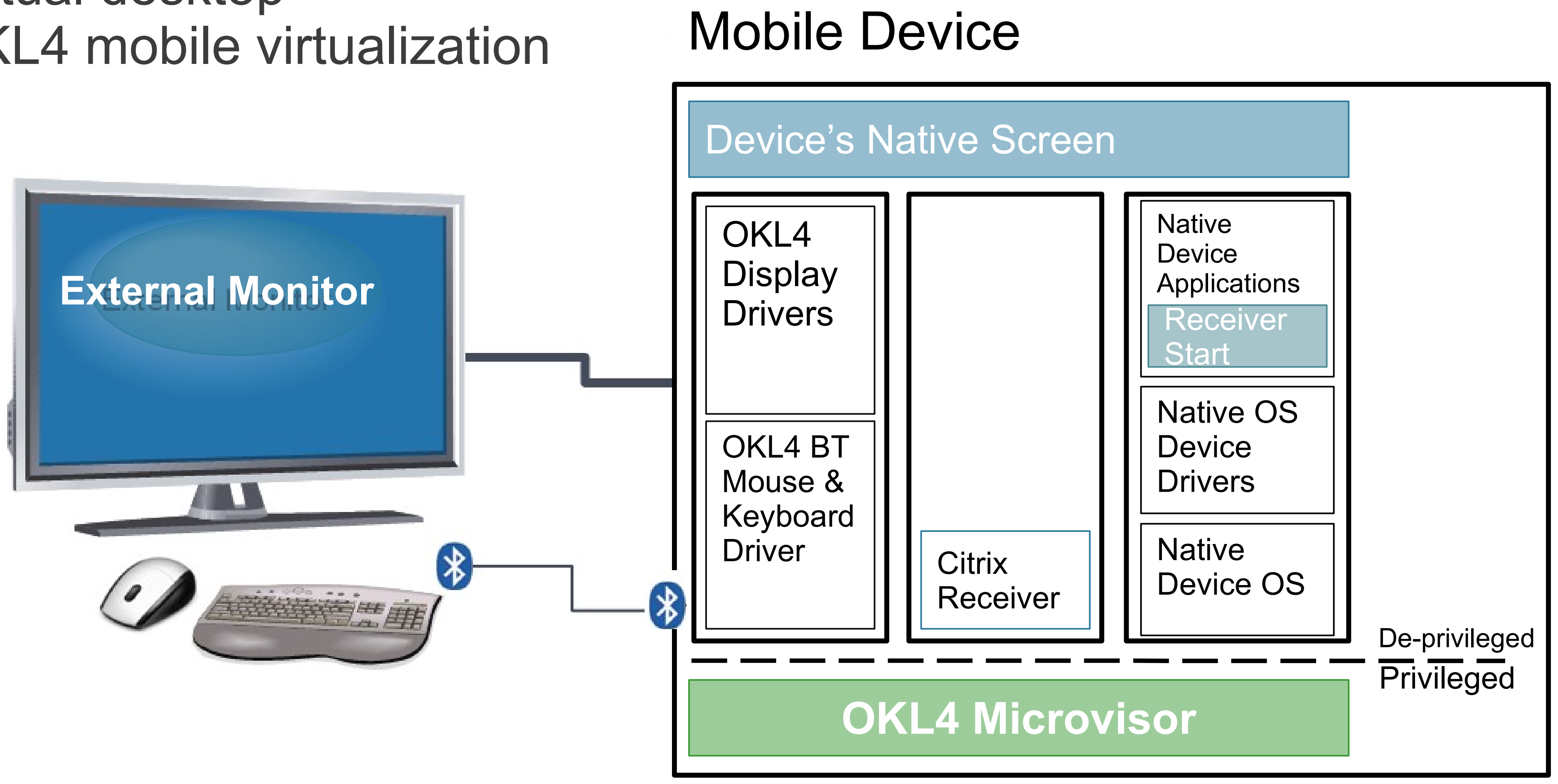
Open Kernel Labs™

CITRIX®



Nirvana Phone

Nirvana phone = Smartphone
+ Full-sized display
+ Keyboard & mouse
+ Virtual desktop
+ OKL4 mobile virtualization

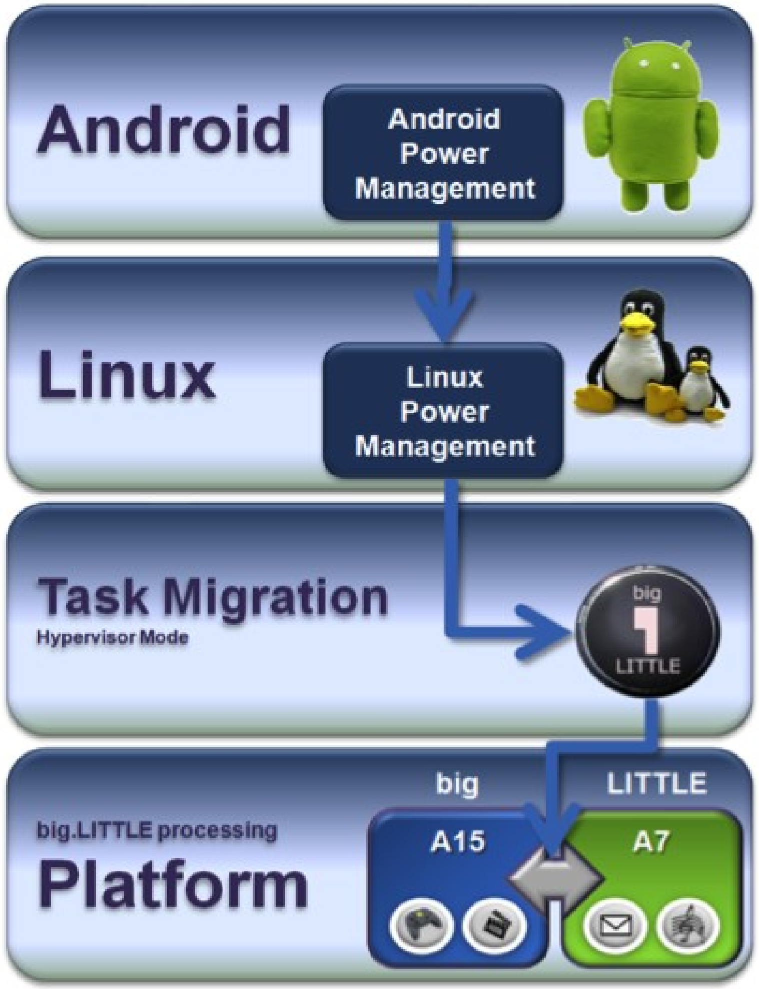


Demo video:
http://www.youtube.com/user/OpenKernelLabs

Use Case: ARM big.LITTLE

http://www.arm.com/products/processors/technologies/bigLITTLEprocessing.php

- Connects the performance of the ARM Cortex-A15 MPCore™ processor with the energy efficiency of the Cortex-A7 processor, enabling the same application software to switch seamlessly between them.
- By selecting the optimum processor for each task, big.LITTLE can extend battery life by up to 70%.
  - For both server and mobile!
- ARM has proposed two usage models, task migration and MP.
- Hardware Supported Virtualization



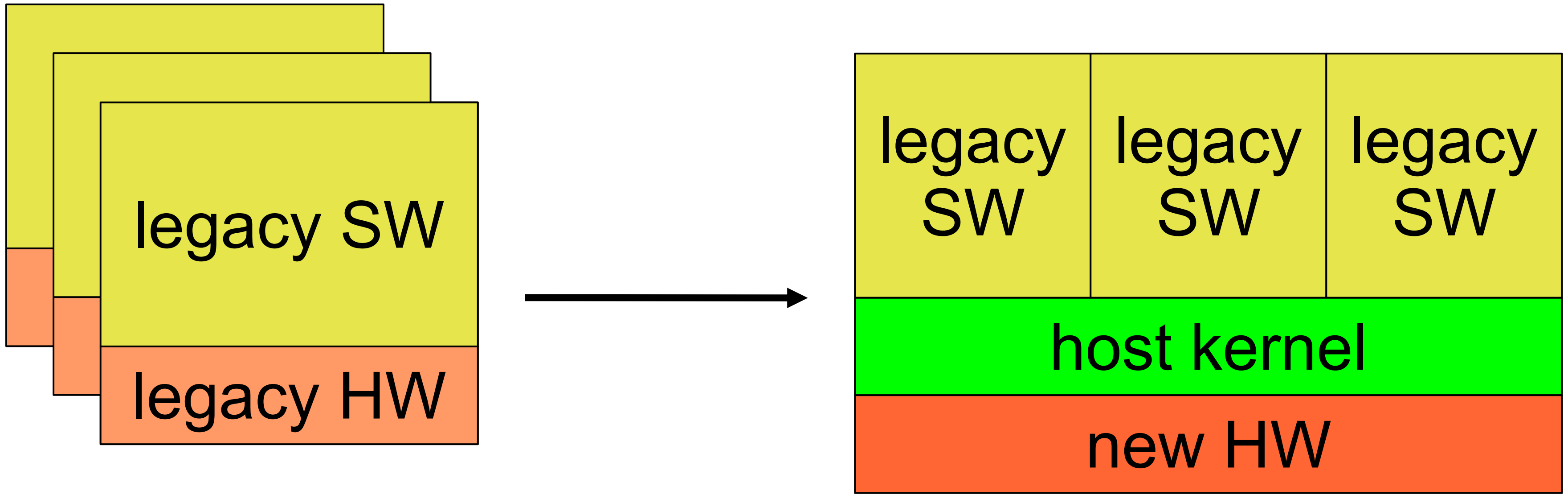
Video: http://www.youtube.com/watch?v=DZbKrGYGnT0

Embedded Virtualization Use Case

- Workload consolidation
- Legacy software
- Multicore enablement
- Improve reliability
- Secure monitoring

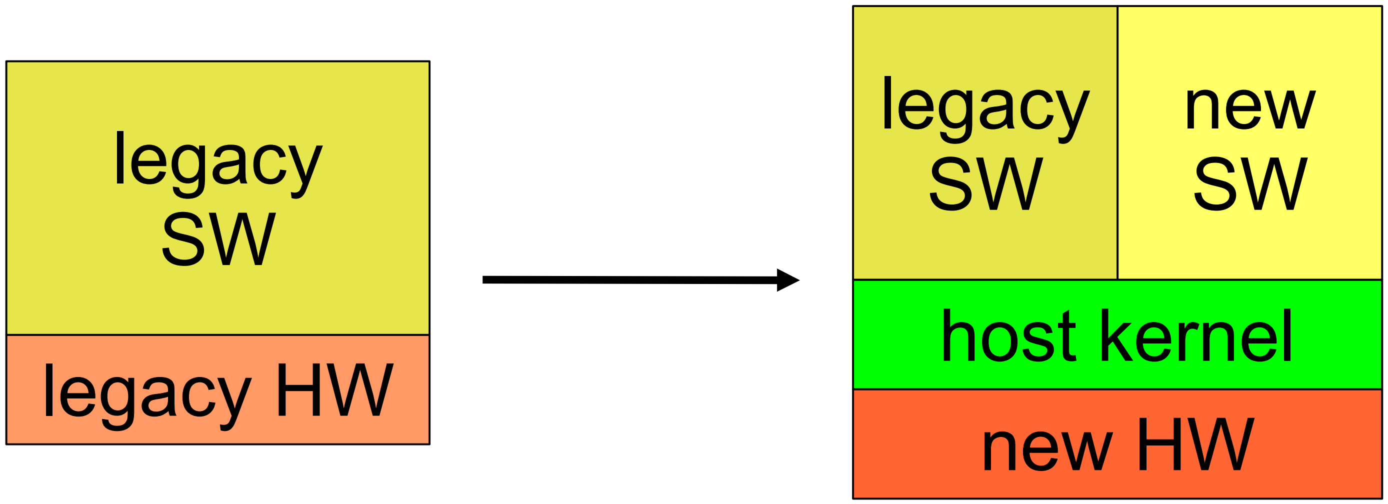
Use Case: Workload Consolidation

- Consolidate legacy systems

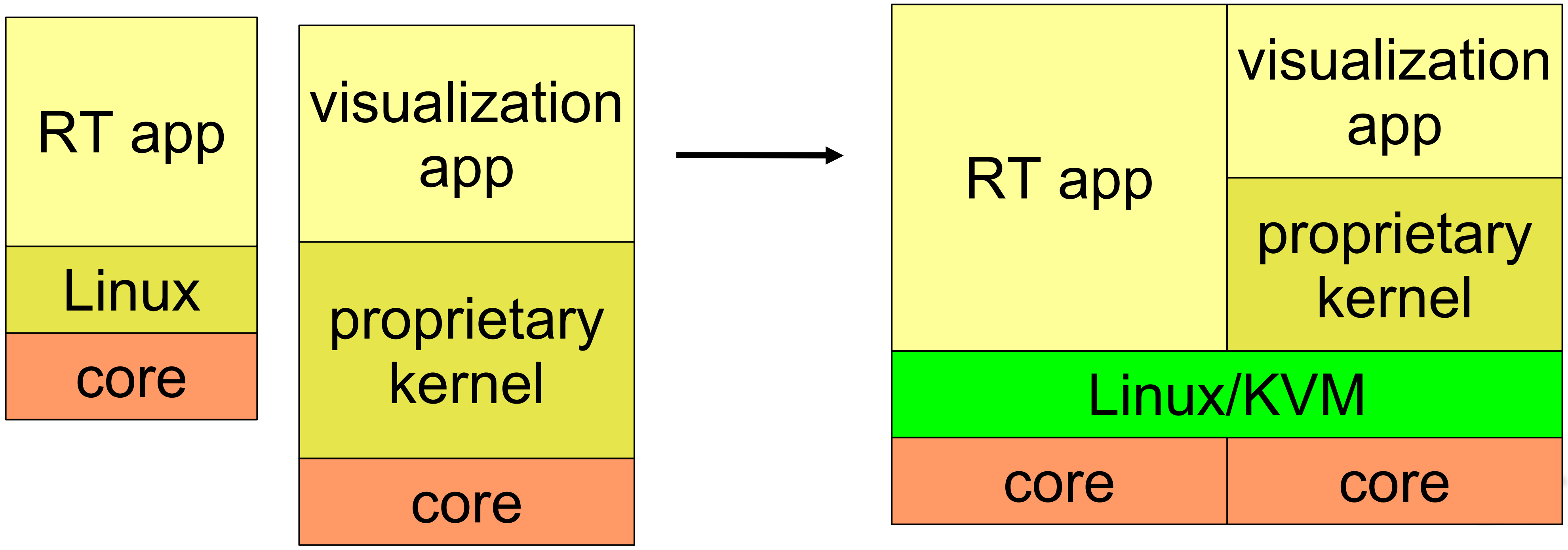


Use Case: Legacy Software

- Run legacy software on new core/chip/board with full virtualization

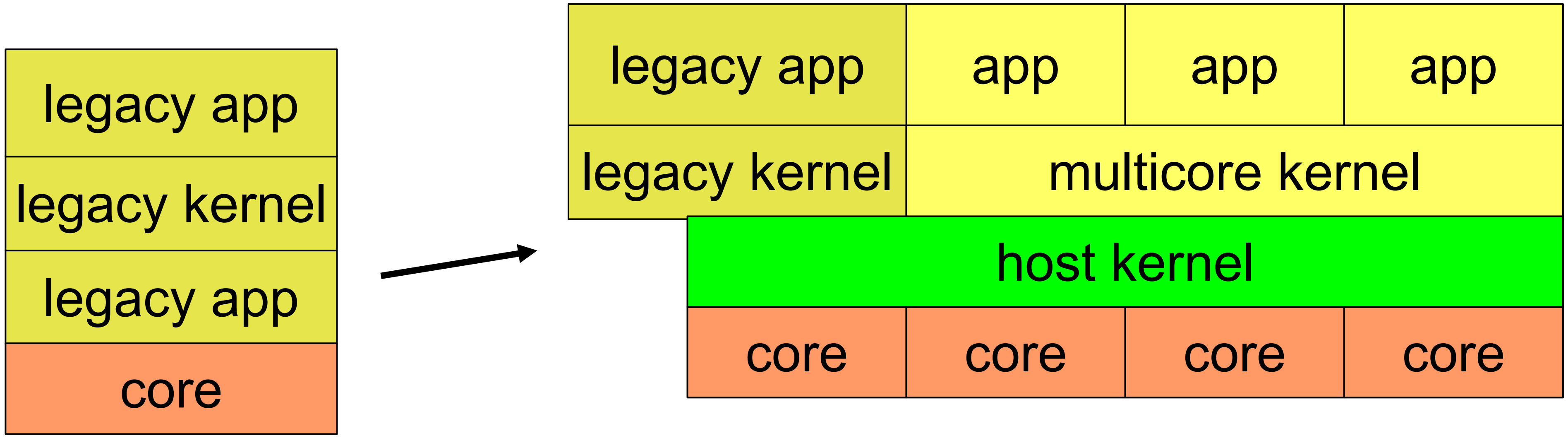


- Consolidate legacy software

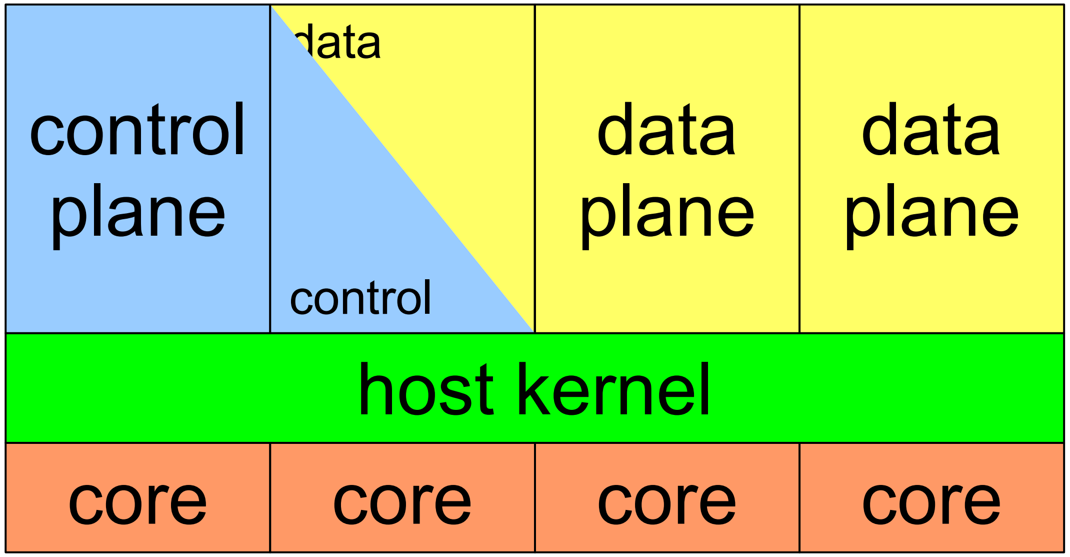


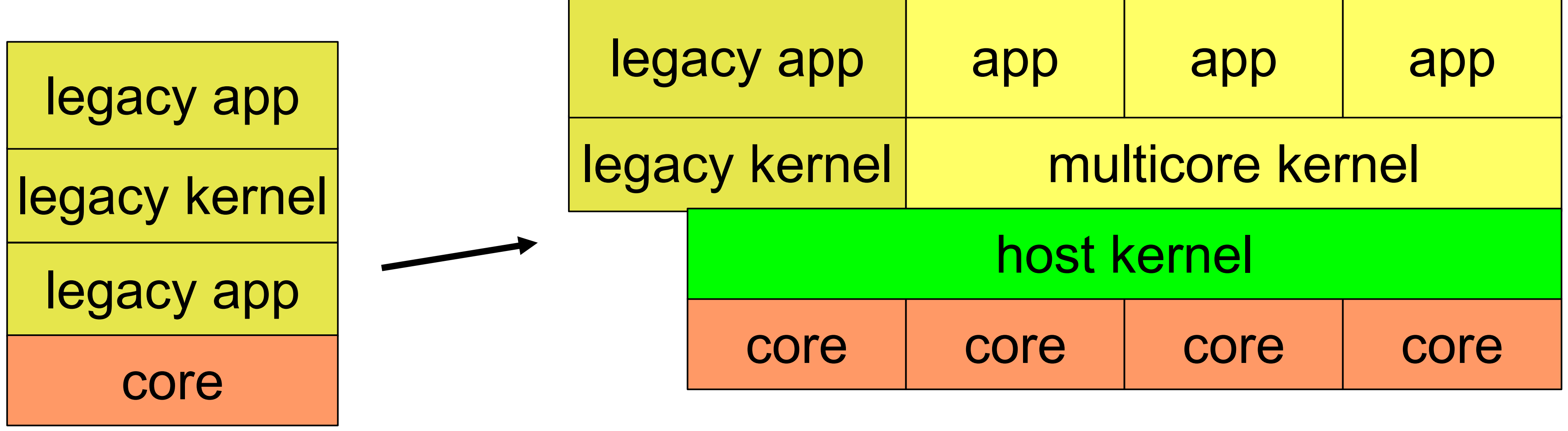
Use Case: Multicore Enablement

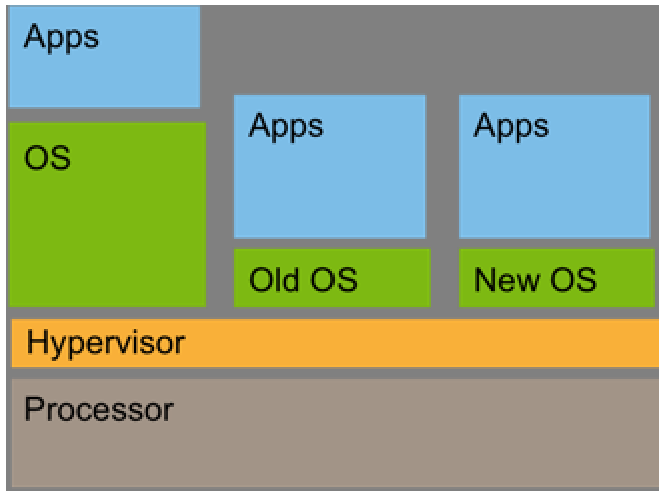
- Legacy uniprocessor applications

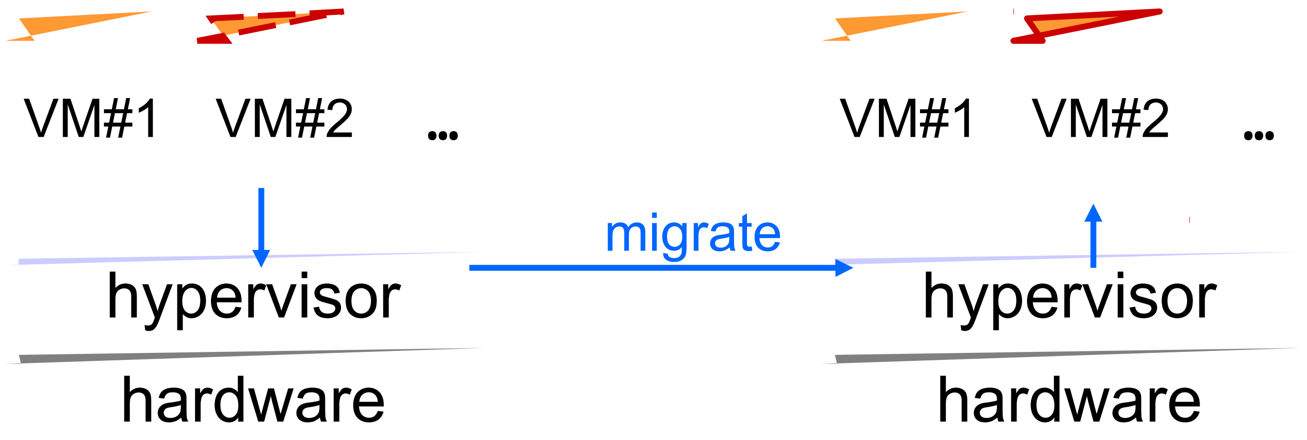


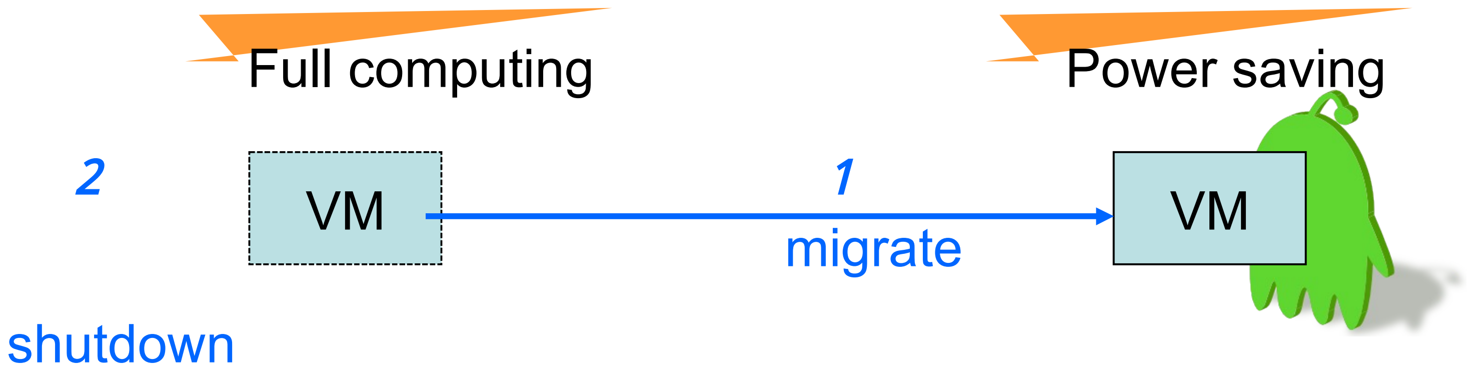
- Flexible resource management




legacy app
legacy kernel
legacy app
core
legacy app
app
app
app
legacy kernel
multicore kernel
host kernel
core
core
core
core

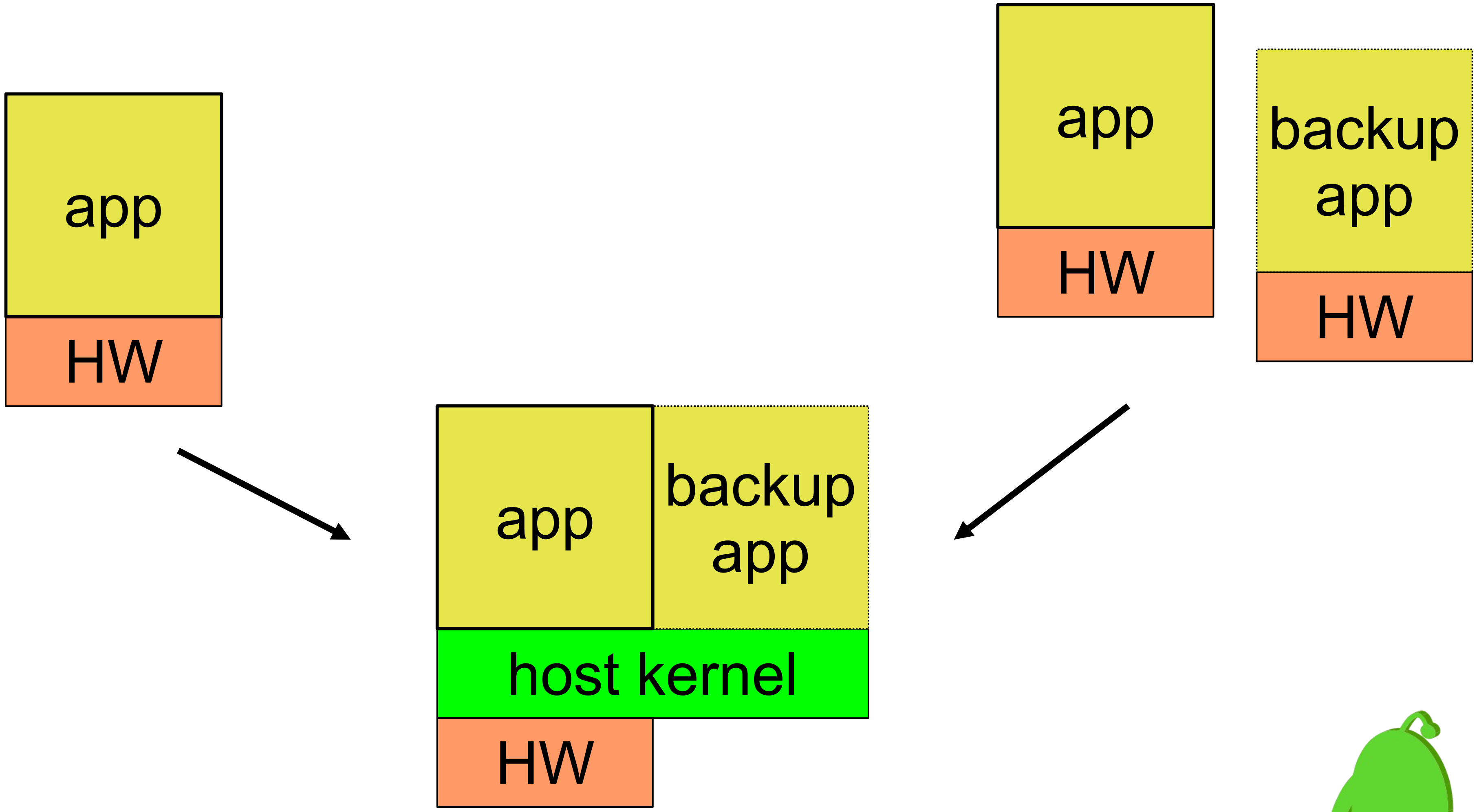

Apps
OS
Apps
Apps
Old OS
New OS
Hypervisor
Processor


VM#1
VM#2
...
hypervisor
hardware
migrate
VM#1
VM#2
...
hypervisor
hardware


Full computing
Power saving
2
VM
1
migrate
VM
shutdown

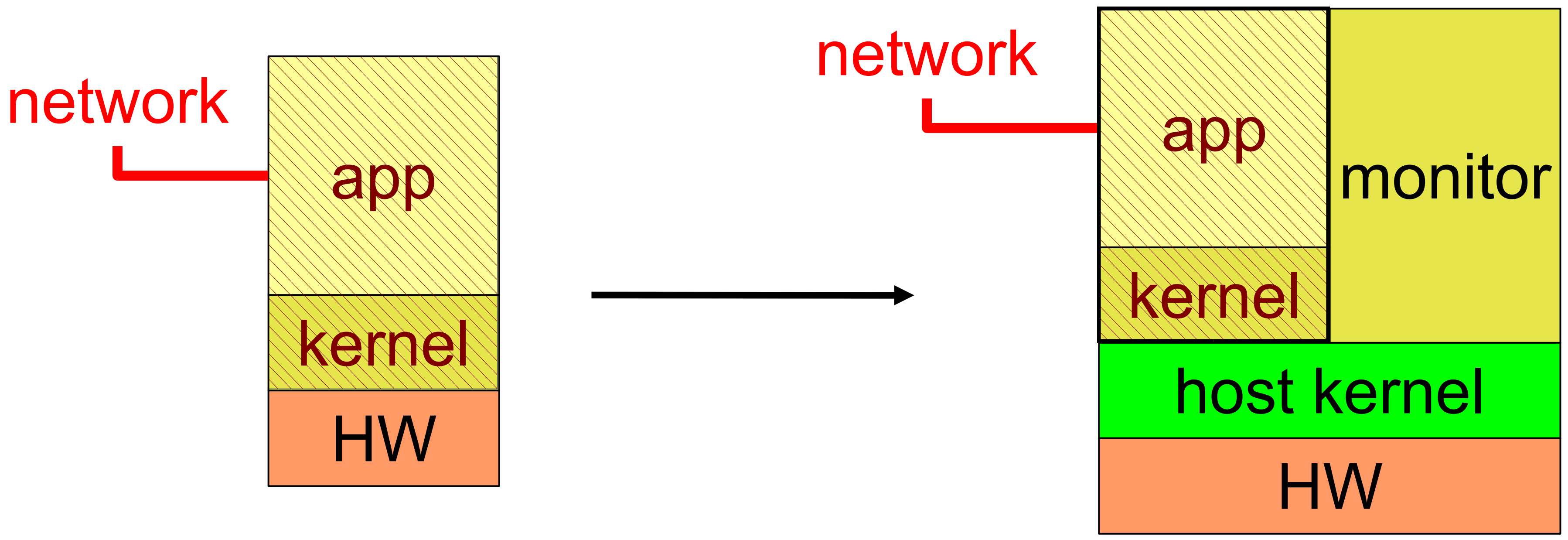
Use Case: Improved Reliability

- Hot standby without additional hardware



Use Case: Secure Monitoring

- Protect monitoring software



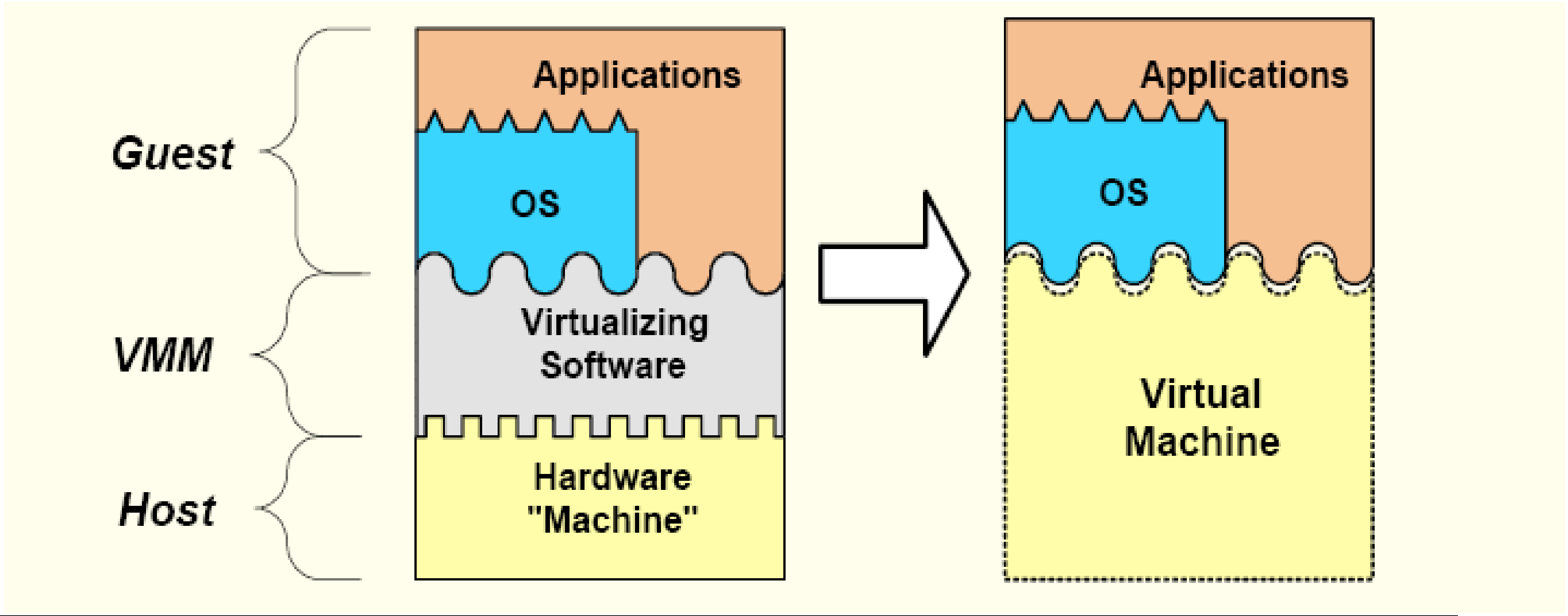
ARM Virtualization

"All problems in computer science can be solved by another level of indirection."

-- David Wheeler --

Virtual Machine

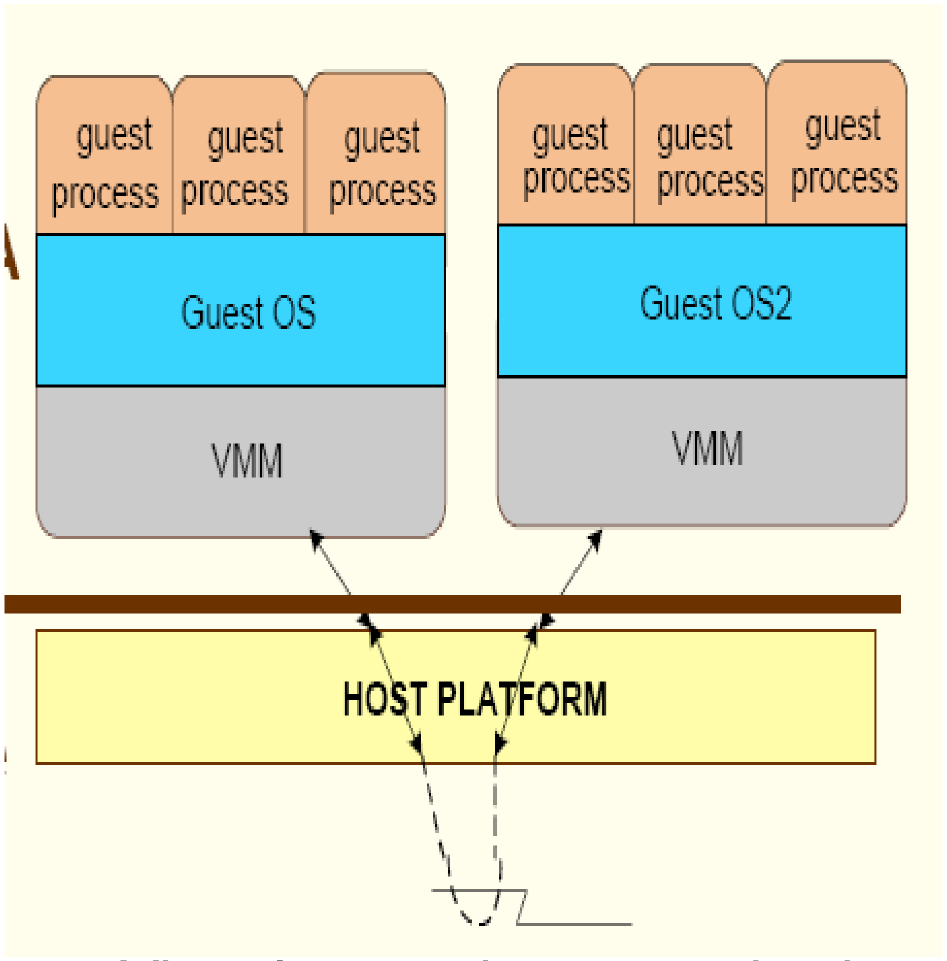
- Gerald Popek and Robert Goldberg defined it as "**efficient, isolated duplicate of a real machine**"
  - Add Virtualizing Software to a Host platform and support Guest process or system on a Virtual Machine (VM)



The software that provides this illusion is the Virtual Machine Monitor (VMM, mostly used synonymous with Hypervisor)

System Virtual Machine

- Provide a system environment
- Constructed at ISA level
- Allow multiple OS environments, or support time sharing.
- virtualizing software that implements system VM is called as VMM (virtual machine monitor)
- Examples:
  - IBM VM/360, VMware, VLX, WindRiver Hypervisor, ENEA Hypervisor
  - Xtratum, Lguest, BhyVe (BSD Hypervisor)
  - **Xen**, **KVM**, **OKL4**, **Xvisor**, **Codezero**

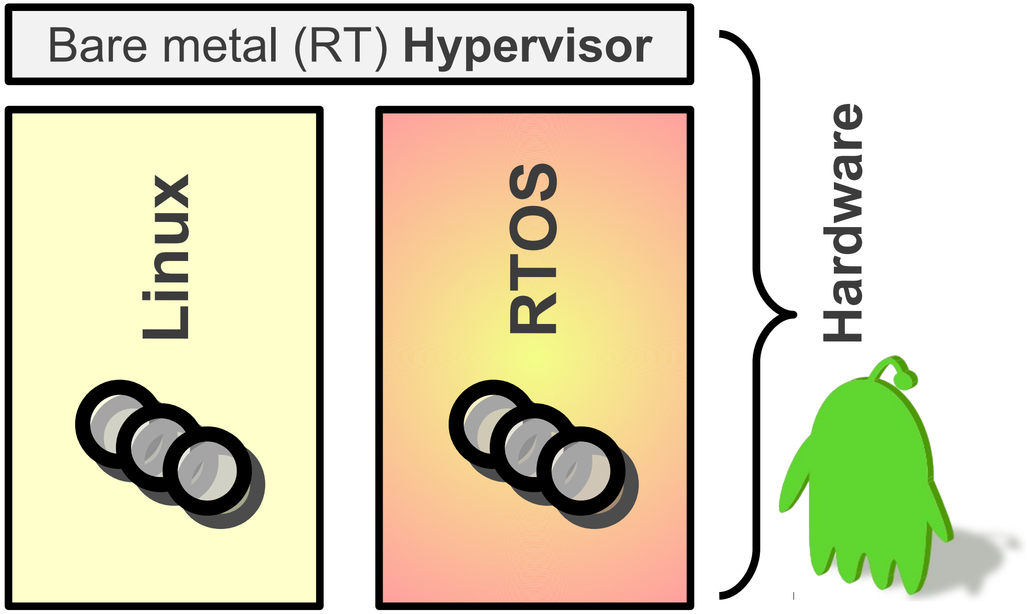


Virtual network communication

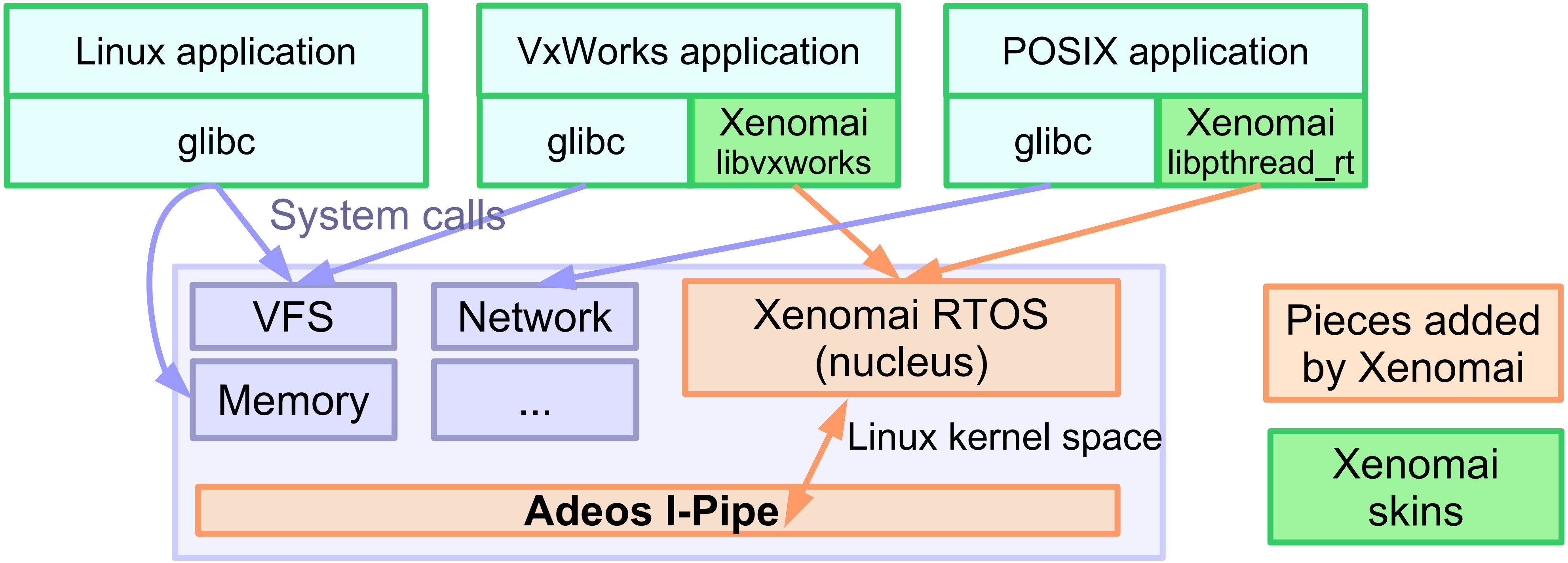
NOTE: We only focus on system virtual machine here.
Therefore, this presentation ignores Linux vserver, FreeBSD jail, etc.

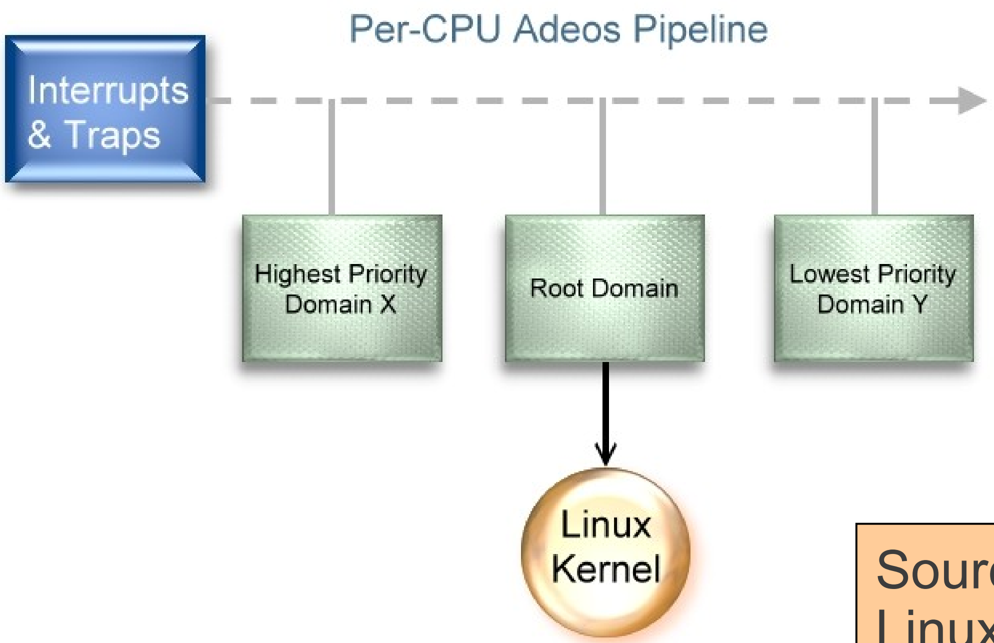
Virtualization is Common Technique

- Example: In the past, Linux is far from being real-time, but RTLinux/RTAI/Xenomai/Xtratum attempted to "improve" Linux by introducing new virtualization layer.
- real-time capable virtualization
- Dual kernel approach



Example: Xenomai (Linux Realtime Extension)

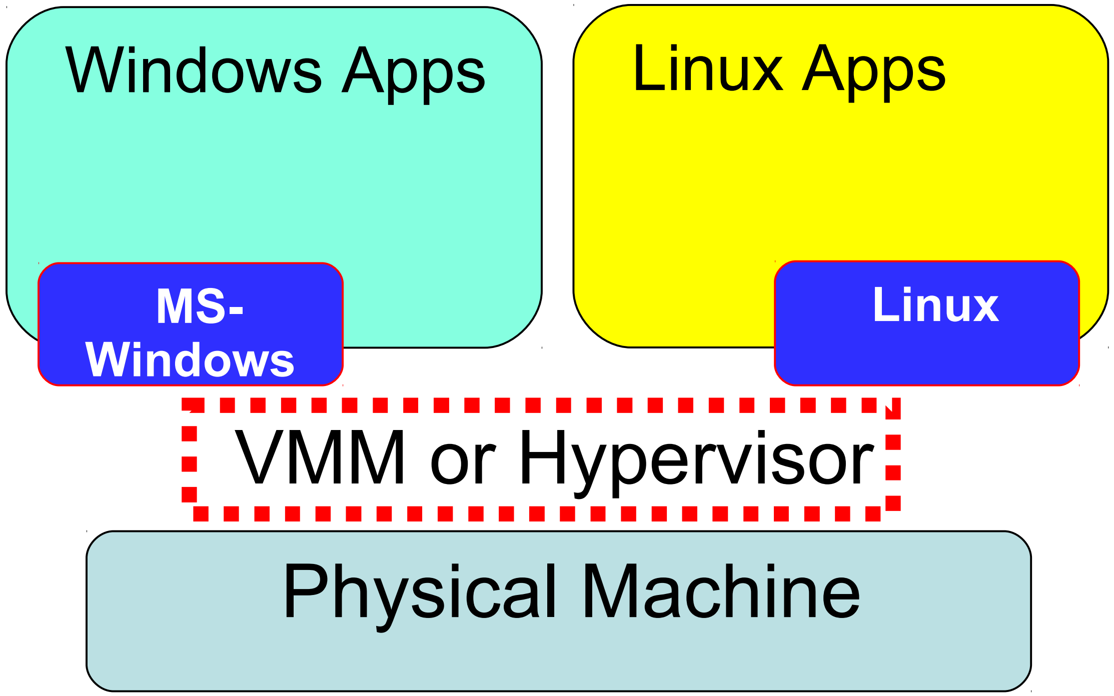




- From Adeos point of view, guest OSes are prioritized domains.
- For each event (interrupts, exceptions, syscalls, etc...), the various domains may handle the event or pass it down the pipeline.

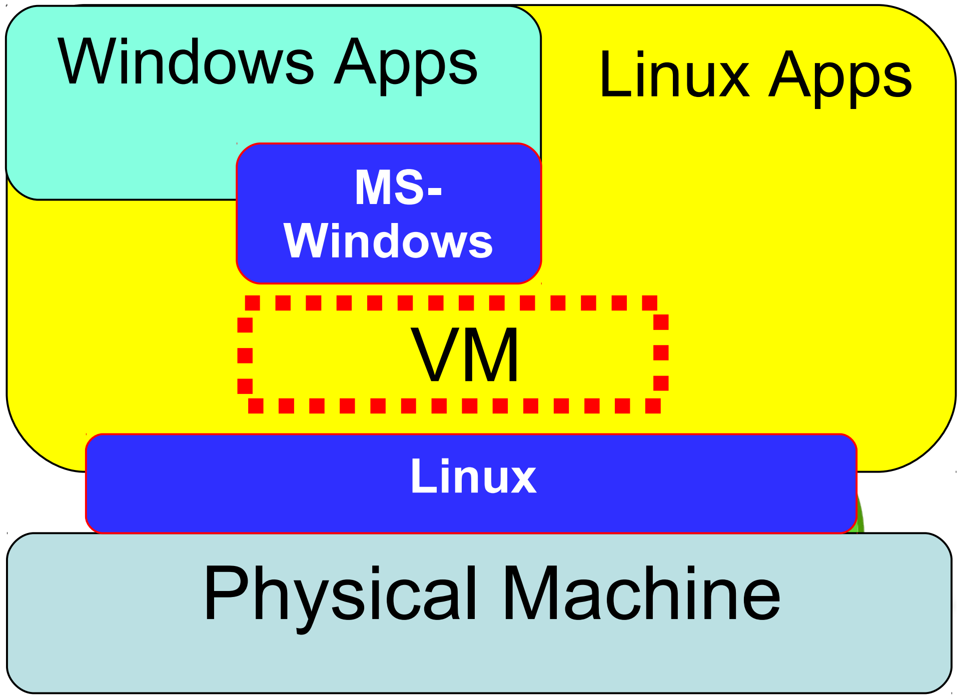
Source: Real-time in embedded Linux systems, Free Electrons (2011)

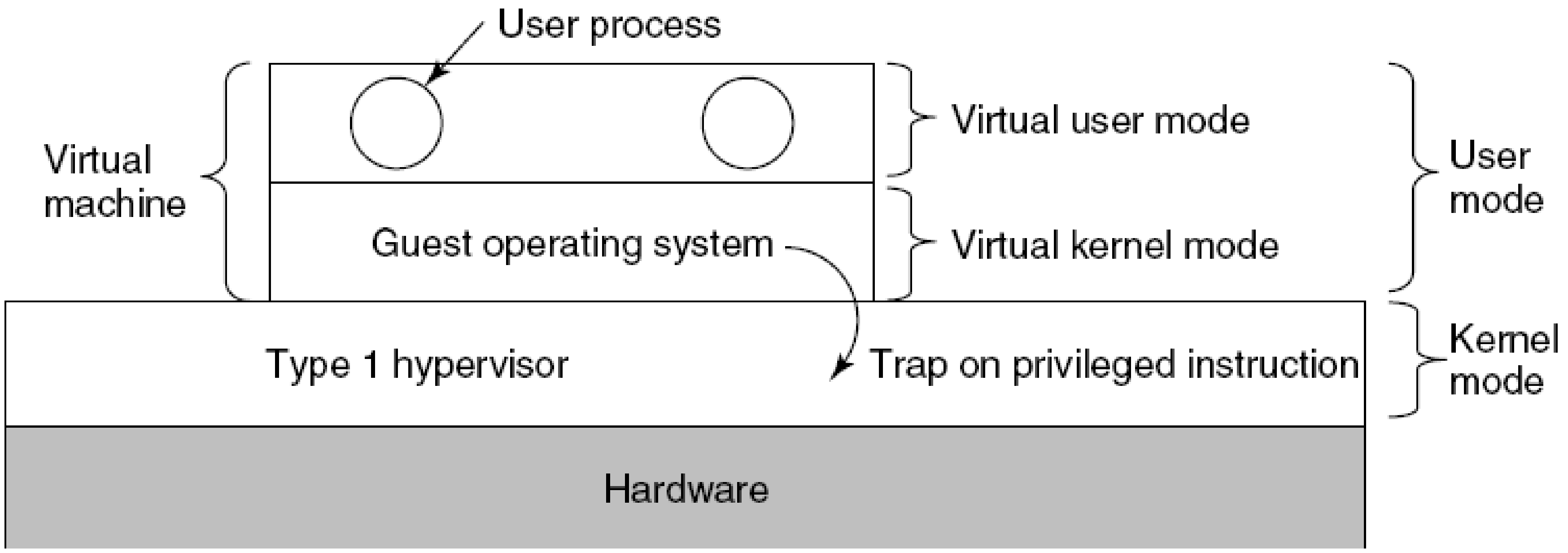
- Type I
- Bare metal system VM

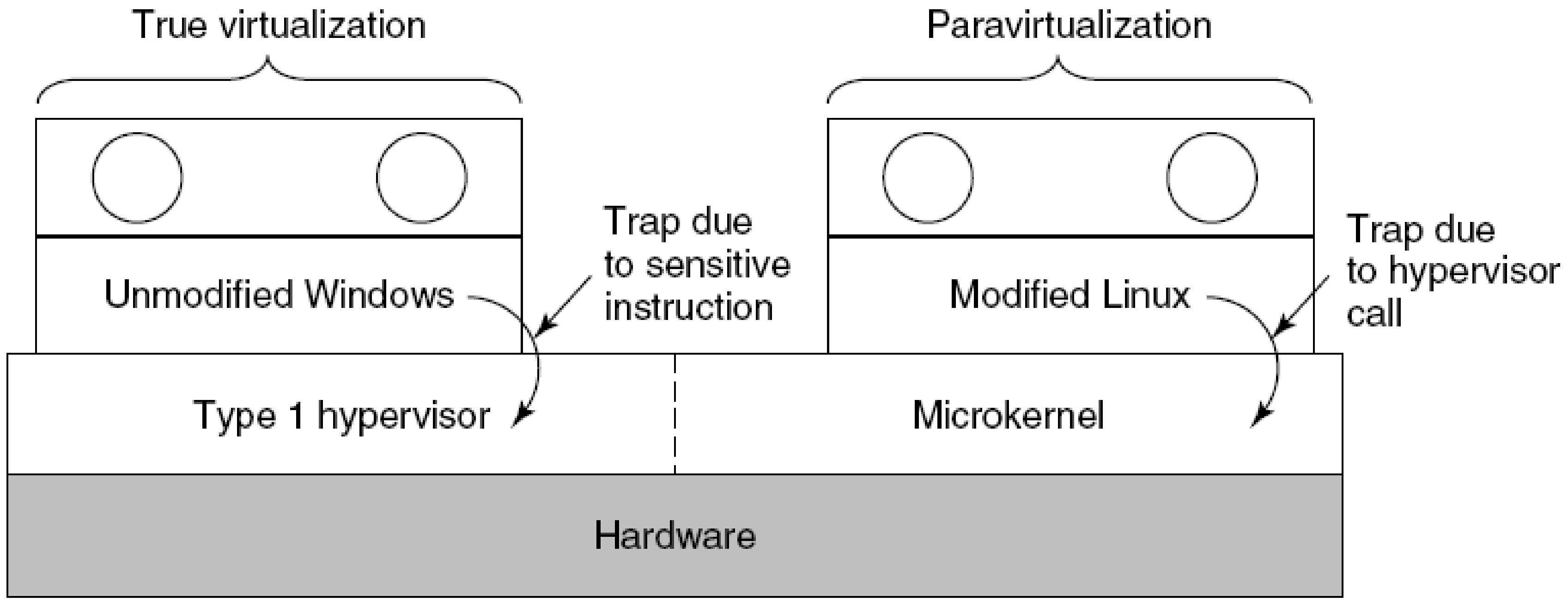


General Classification
of
Virtualization technologies

- Type 2
- Hosted System VM







Source: Operating Systems Design – Lecture 21 Virtualization, Roy Campbell (Fall 2010)

Virtualizable

is a property of the Instruction Set Architecture (ISA)

- A **sensitive instruction**
  - changes the configuration or mode of the processor,

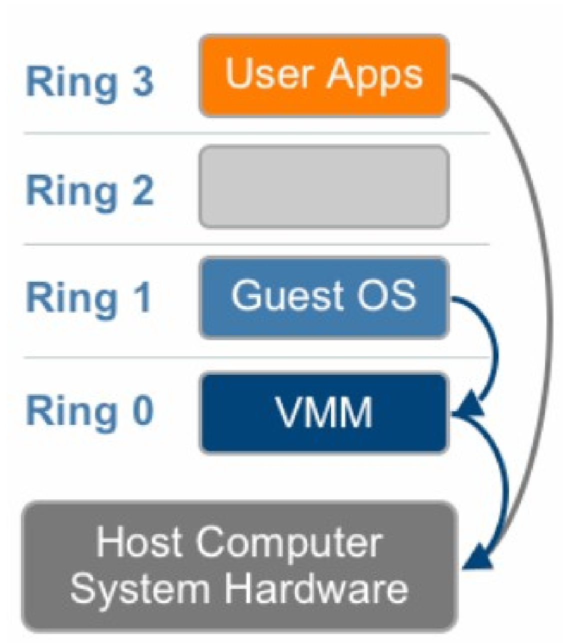
  or
  - depends on its behavior of the processor's state

- A **privileged instruction**
  - must be executed with sufficient privilege
  - causes a trap in user mode

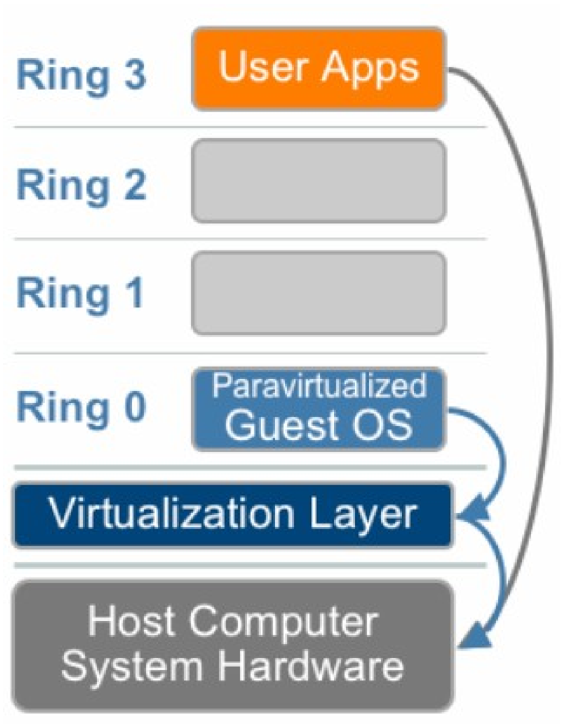
If all sensitive instructions are privileged, a VMM can be written.

System Virtualization Implementations

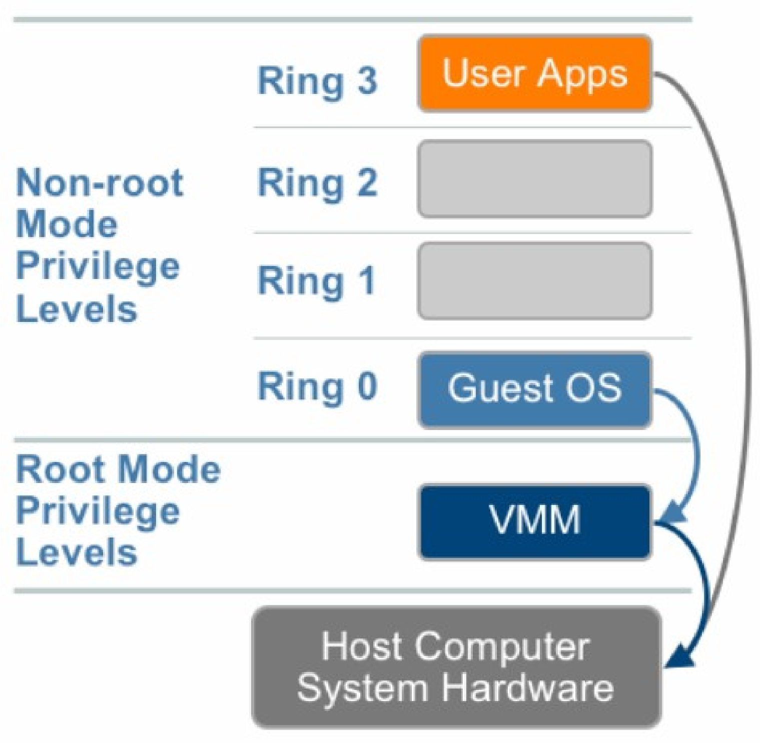
Full Virtualization



Para Virtualization



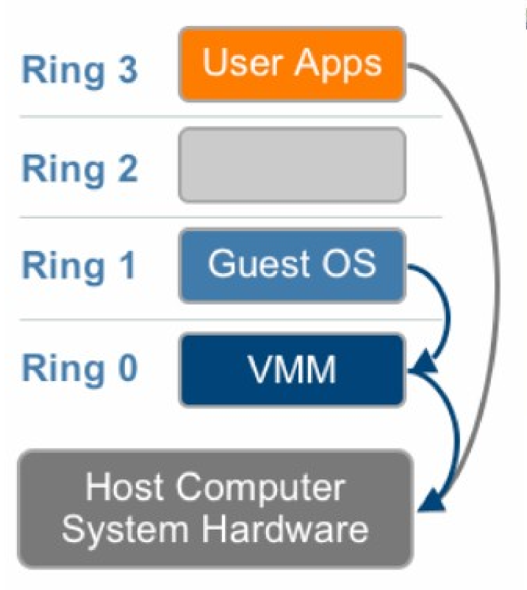
Hardware Assisted Virtualization

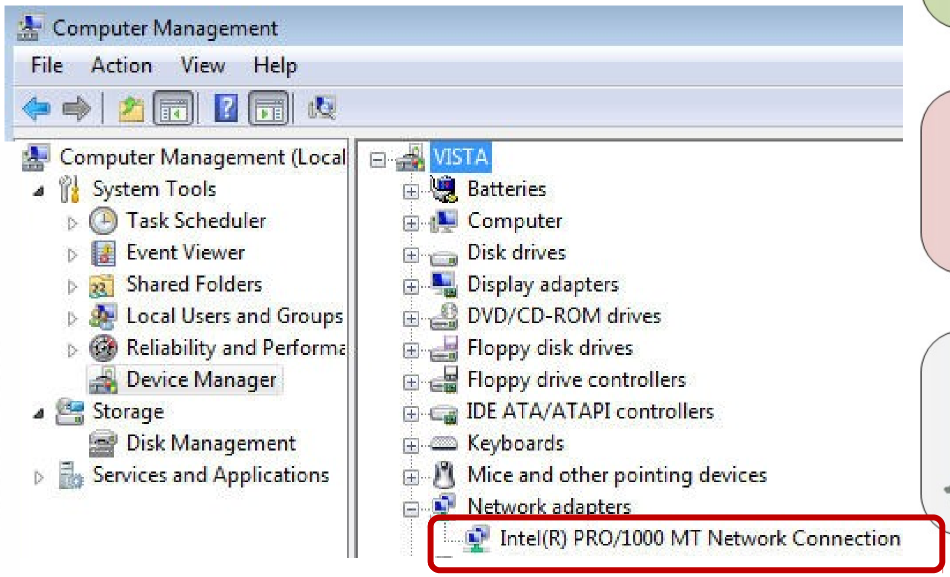


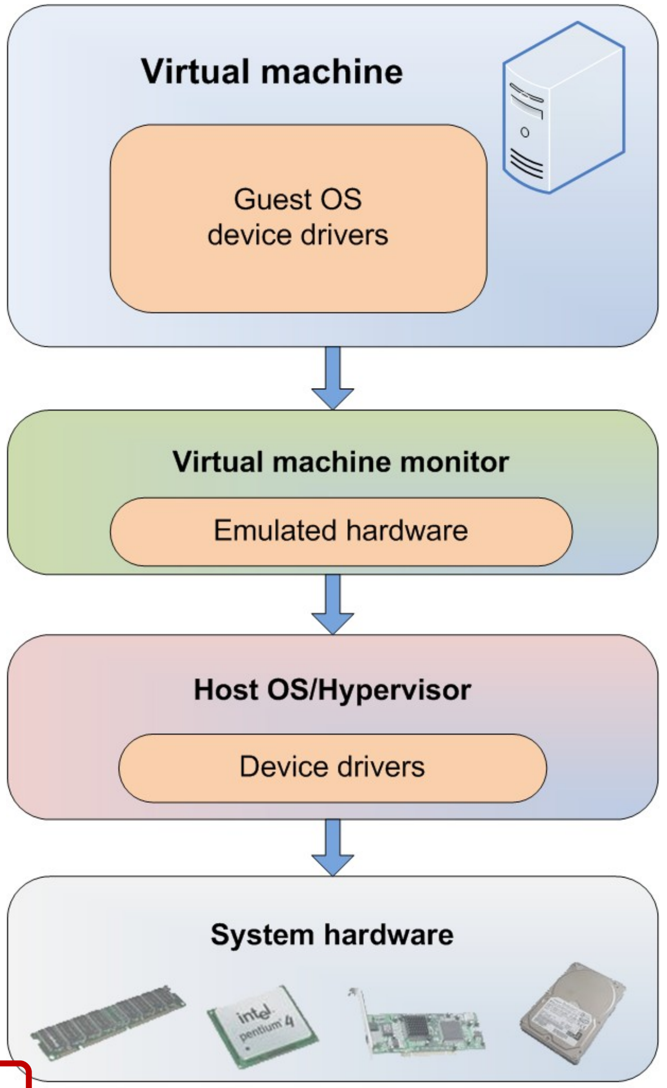
Non-root Mode Privilege Levels
Ring 3
User Apps
Ring 2
Ring 1
Ring 0
Guest OS
Root Mode Privilege Levels
VMM
Host Computer System Hardware

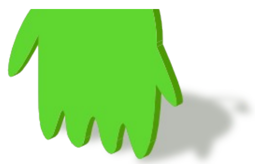
Full Virtualization

- Everything is virtualized
- Full hardware emulation
- Emulation = latency



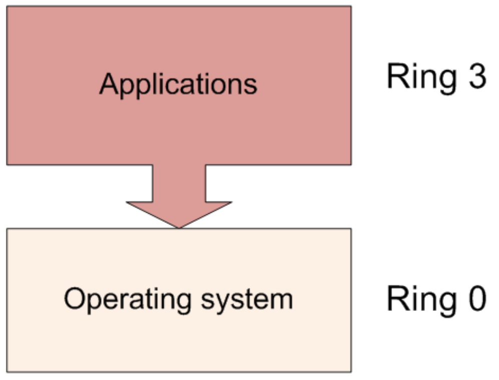




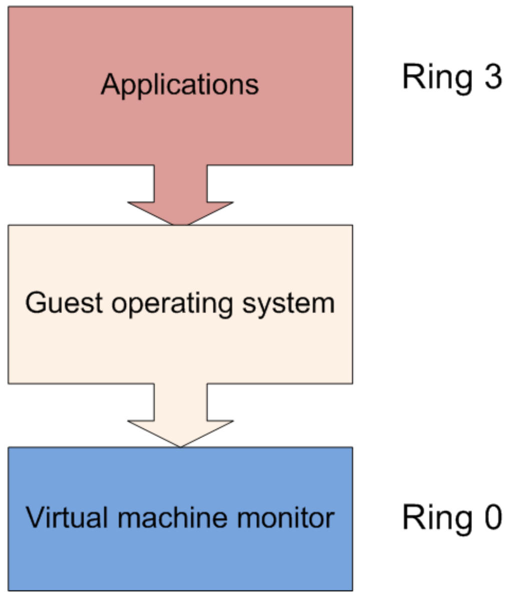


Privileged Instructions

- Privileged instructions: OS kernel and device driver access to system hardware
- **Trapped and Emulated** by VMM
  - execute guest in separate address space in unprivileged mode
  - emulate all instructions that cause traps



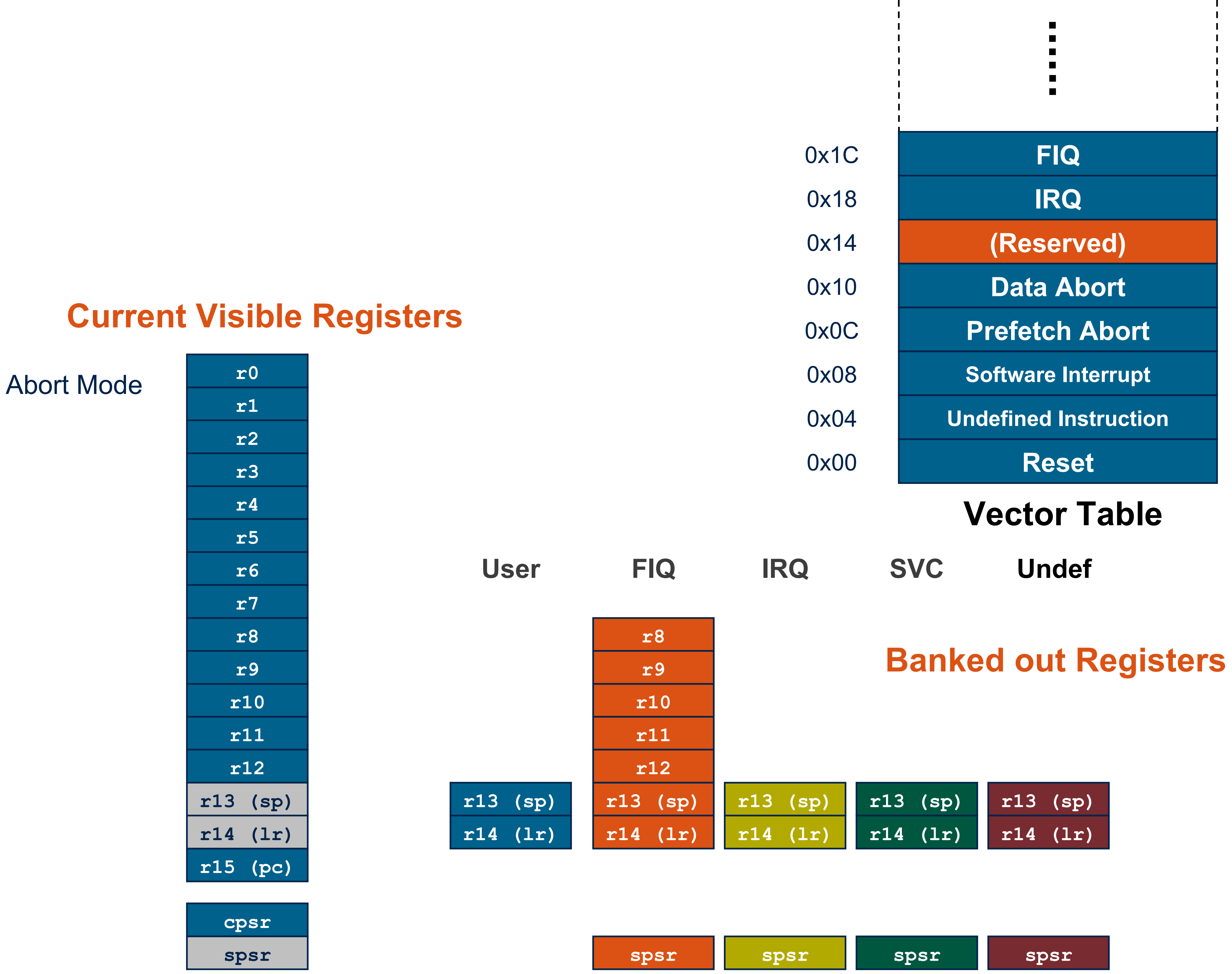
Traditional x86 architecture



Full virtualization

ARM Architecture (armv4)

- 6 basic operating modes (1 user, 5 privileged)
- 37 registers, all 32 bits wide
  - 1 program counter
  - 5 dedicated saved program status registers
  - 1 Current program status register (PSR)
  - 30 general purpose registers
- Special usage
  - r13 (stack pointer)
  - r14 (link register)
  - r15 (program counter, PC)


Current Visible Registers
Abort Mode
r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
cpsr
spsr
0x1C
FIQ
0x18
IRQ
0x14
(Reserved)
0x10
Data Abort
0x0C
Prefetch Abort
0x08
Software Interrupt
0x04
Undefined Instruction
0x00
Reset
Vector Table
User
FIQ
IRQ
SVC
Undef
Banked out Registers
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
spsr

Typical ARM instructions (armv4)

- branch and branch with Link (**`B`**, **`BL`**)
- data processing instructions (**`AND`**, **`TST`**, **`MOV`**, …)
- shifts: logical (**`LSR`**), arithmetic (**`ASR`**), rotate (**`ROR`**)
- test (**`TEQ`**, **`TST`**, **`CMP`**, **`CMN`**)
- processor status register transfer (**`MSR`**, **`MRS`**)
- memory load/store words (**`LDR`**, **`STR`**)
- push/pop Stack Operations (**`STM`**, **`LDM`**)
- software Interrupt (**`SWI`**; operating mode switch)
- co-processor (**`CDP`**, **`LDC`**, **`STC`**, **`MRC`**, **`MCR`**)

Problematic Instructions (1)

- Type 1

  Instructions which executed in user mode will cause **undefined instruction** exception
- Example

  MCR p15, 0, r0, c2, c0, 0

  Move r0 to c2 and c0 in coprocessor specified by p15 (co-processor) for operation according to option 0 and 0
  – MRC: from coproc to register
  – MCR: from register to coproc
- Problem:
  – Operand-dependent operation

Source: 前瞻資訊科技 – 虛擬化，薛智文，台大資訊所 (2011)

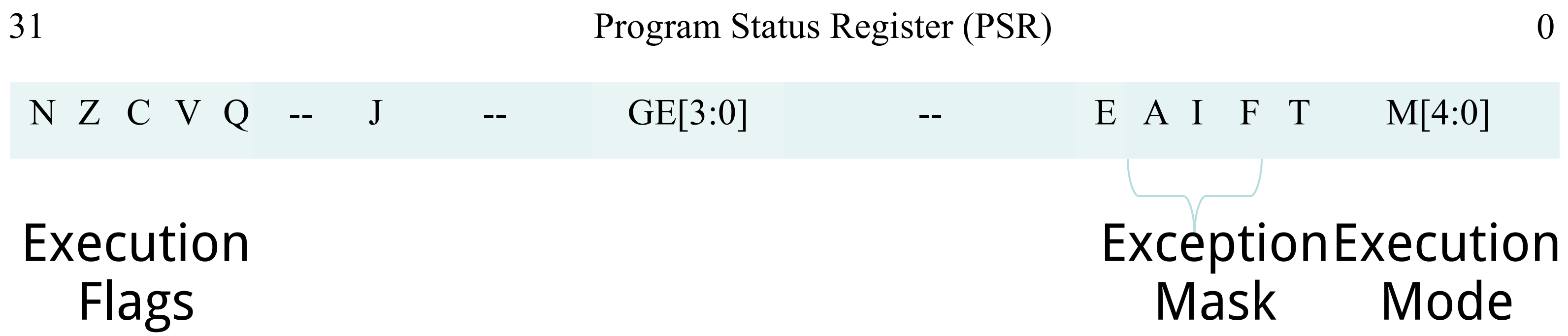
Problematic Instructions (2)

- Type 2

  Instructions which executed in user mode will have **no effect**
- Example

  MSR cpsr_c, #0xD3

  Switch to privileged mode and disable interrupt


31
Program Status Register (PSR)
0
N Z C V Q -- J -- GE[3:0] -- E A I F T M[4:0]
Execution Flags
Exception Mask
Execution Mode

Problematic Instructions (3)

- Type 3

  Instructions which executed in user mode will cause **unpredictable behaviors**.
- Example

  ```
  MOVS     PC, LR
  ```

  The return instruction

  changes the **program counter** and switches to **user mode**.
- This instruction causes unpredictable behavior when executed in user mode.

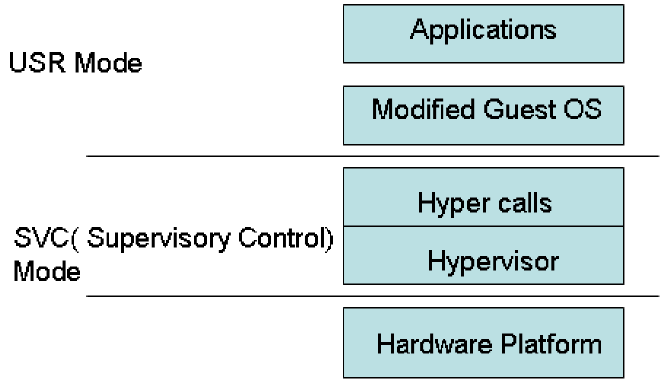
ARM Sensitive Instructions

- Coprocessor Access Instructions
  `MRC` / **`MCR`** / **`CDP`** / **`LDC`** / **`STC`**
- SIMD/VFP System Register Access Instructions
  `VMRS` / **`VMSR`**
- TrustZone Secure State Entry Instructions
  `SMC`
- Memory-Mapped I/O Access Instructions
  Load/Store instructions from/into memory-mapped I/O locations
- Direct (Explicit/Implicit) CPSR Access Instructions
  `MRS` / **`MSR`** / **`CPS`** / **`SRS`** / **`RFE`** / **`LDM`** (conditional execution) / **`DPSPC`**
- Indirect CPSR Access Instructions
  `LDRT` / **`STRT`** – Load/Store Unprivileged ("As User")
- Banked Register Access Instructions
  `LDM`/**`STM`** (User mode registers)

Solutions to Problematic Instructions

[**Hardware** Techniques]

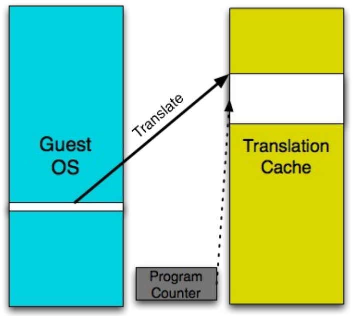
- Privileged Instruction Semantics dictated/translated by instruction set architecture
- MMU-enforced traps
  - Example: page fault
- Tracing/debug support
  - Example: **`bkpt`** (breakpoint)
- Hardware-assisted Virtualization
  - Example: extra privileged mode, HYP, in ARM Cortex-A15



Solutions to Problematic Instructions

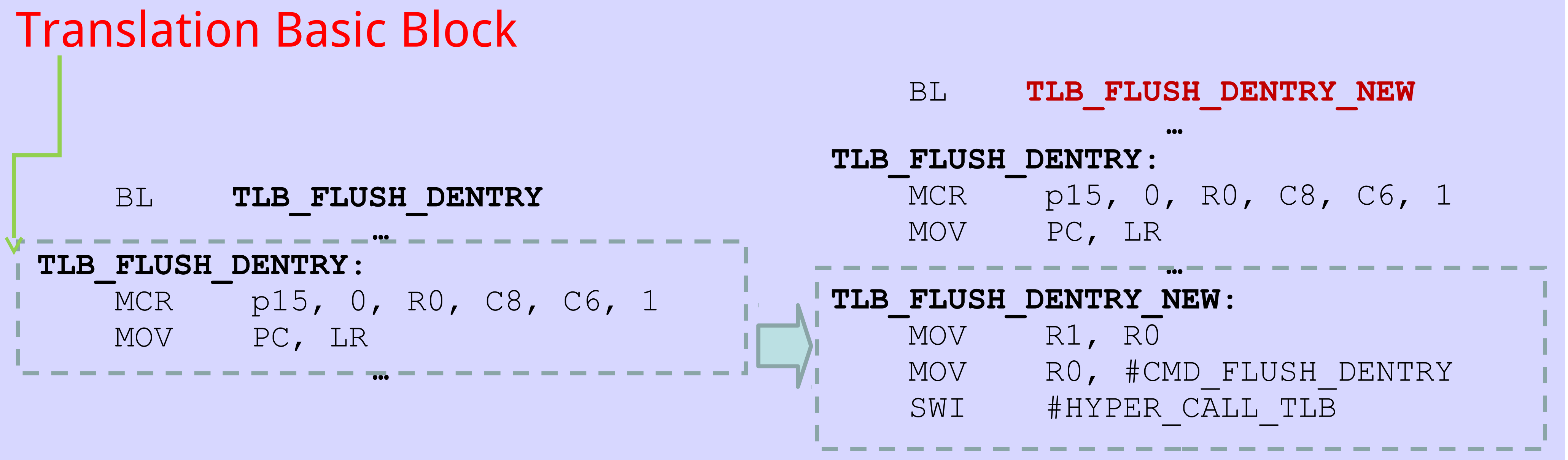
[Software Techniques]

Complexity	**Binary translation**	**Hypercall**
Design	**High**	**Low**
Implementation	**Medium**	**High**
Runtime	**High**	**Medium**
Mapped to programming languages	Virtual function	Normal function



Method: trap and emulate

Dynamic Binary Translation

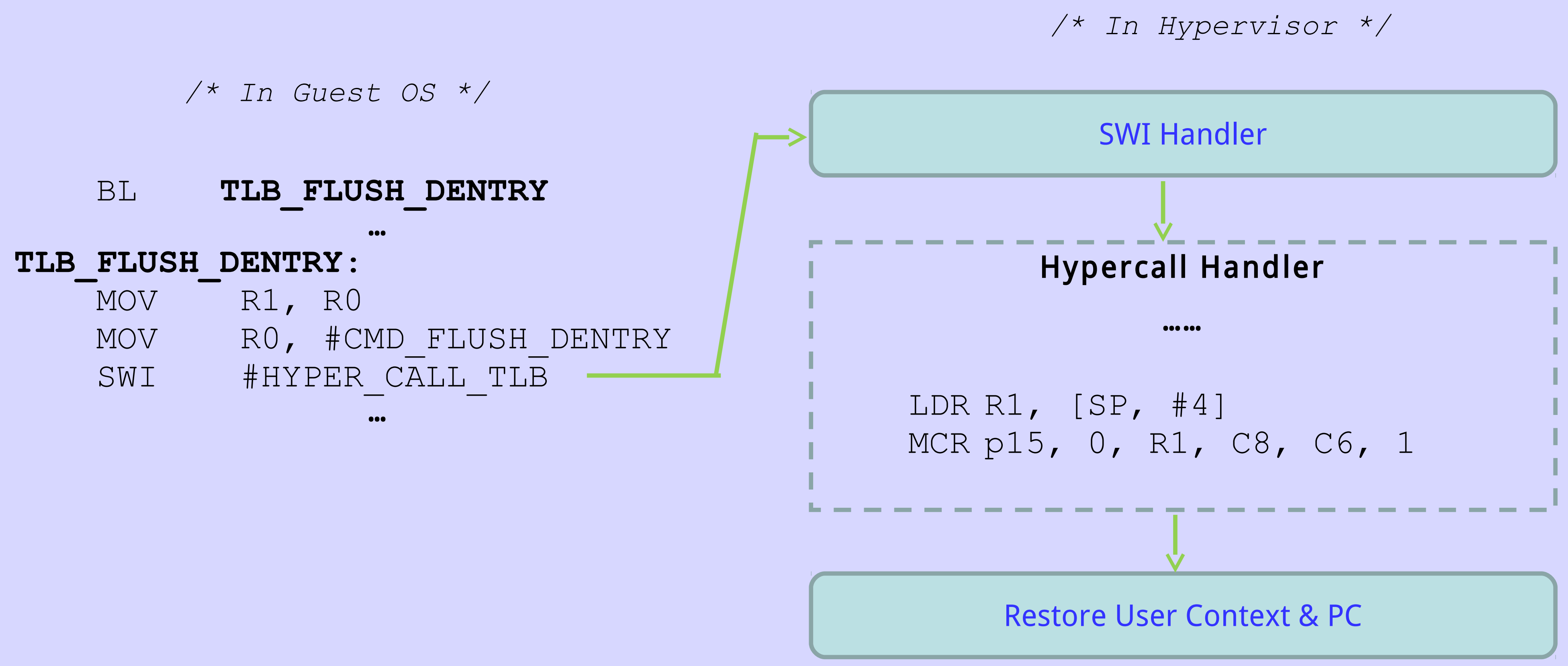


```
    BL     TLB_FLUSH_DENTRY
                    …
TLB_FLUSH_DENTRY:
    MCR    p15, 0, R0, C8, C6, 1
    MOV    PC, LR
                    …
```

```
    BL     TLB_FLUSH_DENTRY_NEW
                    …
TLB_FLUSH_DENTRY:
    MCR    p15, 0, R0, C8, C6, 1
    MOV    PC, LR
                    …
TLB_FLUSH_DENTRY_NEW:
    MOV    R1, R0
    MOV    R0, #CMD_FLUSH_DENTRY
    SWI    #HYPER_CALL_TLB
```

- ARM has a fixed instruction size
  - 32-bit in ARM mode and 16-bit in Thumb mode
- Perform binary translation
  - Follow control-flow
  - Translate basic block (if not already translated) at the current PC
  - Ensure interposition at end of translated sequence
  - All writes (but not reads) to PC now become problematic instructions
  - Replace problematic instructions 1-1 with hypercalls to trap and emulate → self-modifying code

Virtualization APIs – hypercalls



```
/* In Guest OS */

    BL      TLB_FLUSH_DENTRY
                      …
TLB_FLUSH_DENTRY:
    MOV     R1, R0
    MOV     R0, #CMD_FLUSH_DENTRY
    SWI     #HYPER_CALL_TLB
                      …
```

```
/* In Hypervisor */
```

```
……

LDR R1, [SP, #4]
MCR p15, 0, R1, C8, C6, 1
```

- Use trap instruction to issue hypercall
- Encode hypercall type and original instruction bits in hypercall hint
- Upon trapping into the VMM, decode the hypercall type and the original instruction bits, and emulate instruction semantics

```
mrs Rd, R <cpsr/spsr>
```

cond	0001	0R00	SBO.	-Rd-	SBZ.	0000	SBZ.

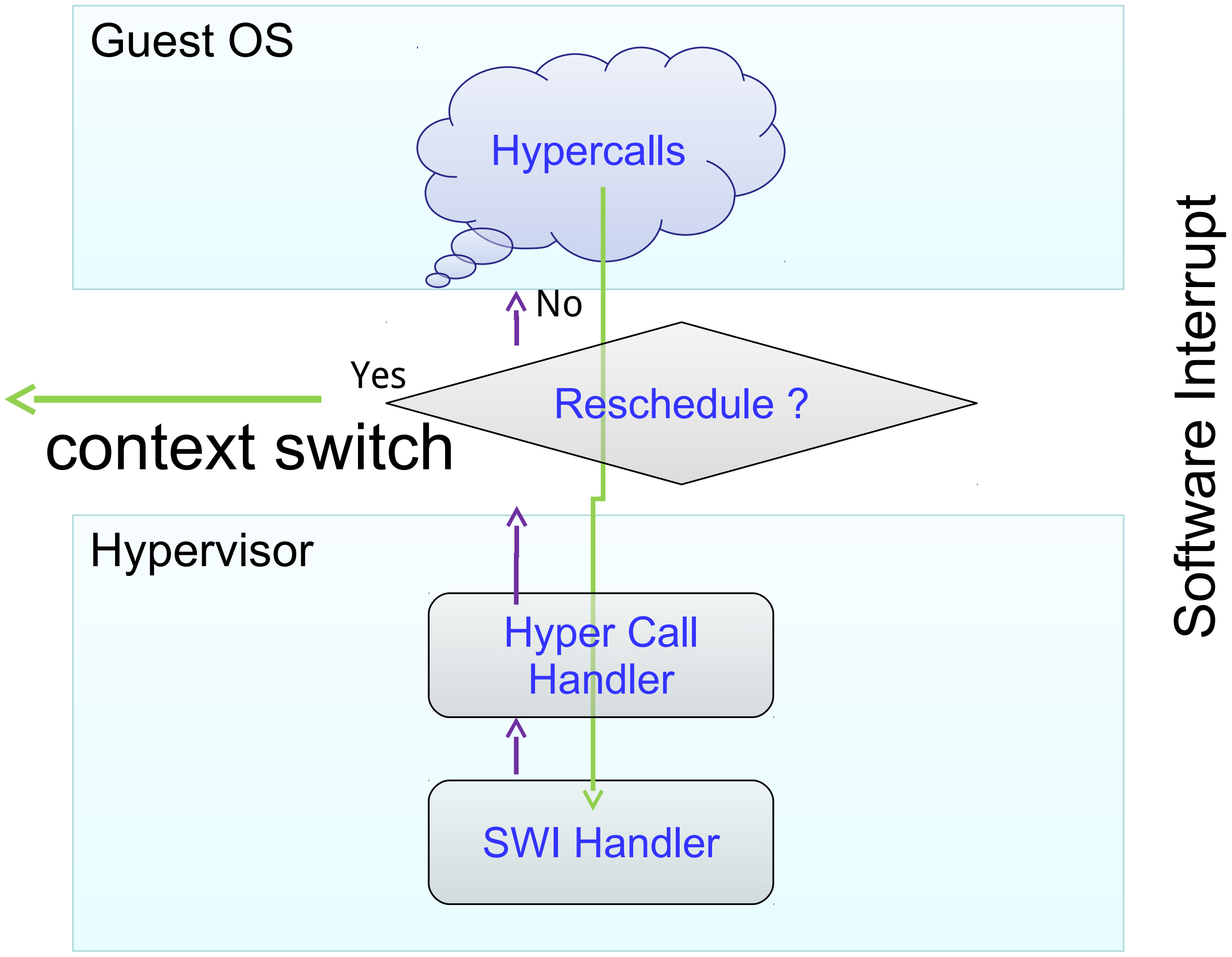
$\Rightarrow$

cond	1111	000010	0R	-Rd-	0000	0000	0000

```
mrs r8, cpsr
```

```
swi 0x088000
```

Hypercall


Guest OS
Hypercalls
No
Yes
Reschedule ?
context switch
Hypervisor
Hyper Call Handler
SWI Handler
Software Interrupt

Case study: Xvisor-ARM

https://github.com/xvisor

- File: arch/arm/cpu/arm32/elf2cpatch.py
  - Script to generate cpatch script from guest OS ELF
- Functionality before generating the final ELF image
  - Each sensitive non-priviledged ARM instruction is converted to a hypercall.
  - Hypercall in ARM instruction set is **`svc`** <imm24> instruction.
  - Encode sensitive non-priviledged instructions in <imm24> operand of **`svc`** instruction. (software interrupt)
  - Each encoded instruction will have its own unique inst_id.
  - The inst_field for each encoded sensitive non-priviledged instruction will be diffrent.

How does Xvisor handle problematic instructions like MSR?

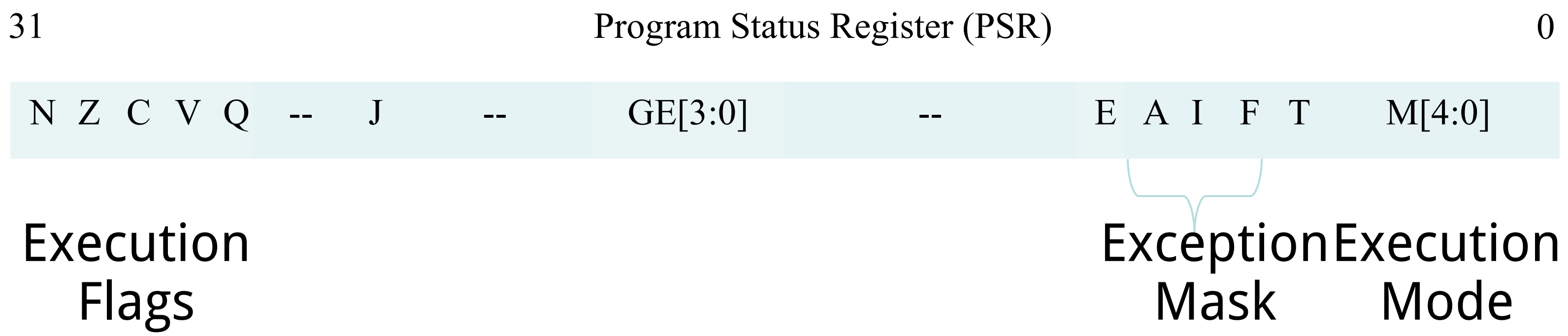
- Type 2

  Instructions which executed in user mode will have **no effect**

- Example

```
MSR cpsr_c, #0xD3
```

Switch to privileged mode and disable interrupt



First, cpatch (ELF patching tool) looks up the instructions…

```
MSR cpsr_c, #0xD3
```

Switch to privileged mode and disable interrupt

```
# MSR (immediate)
#        Syntax:
#                msr<c> <spec_reg>, #<const>
#        Fields:
#                cond = bits[31:28]
#                R = bits[22:22]
#                mask = bits[19:16]
#                imm12 = bits[11:0]
#        Hypercall Fields:
#                inst_cond[31:28] = cond
#                inst_op[27:24] = 0xf
#                inst_id[23:20] = 0
#                inst_subid[19:17] = 2
#                inst_fields[16:13] = mask
#                inst_fields[12:1] = imm12
#                inst_fields[0:0] = R
```

```
# MSR (immediate)
#       Syntax:
#               msr<c> <spec_reg>, #<const>
#       Fields:
#               cond = bits[31:28]
#               R = bits[22:22]
#               mask = bits[19:16]
#               imm12 = bits[11:0]
#       Hypercall Fields:
#               inst_cond[31:28] = cond
#               inst_op[27:24] = 0xf
#               inst_id[23:20] = 0
#               inst_subid[19:17] = 2
#               inst_fields[16:13] = mask
#               inst_fields[12:1] = imm12
#               inst_fields[0:0] = R
```

```
def convert_msr_i_inst(hxstr):
        hx = int(hxstr, 16)
        inst_id = 0
        inst_subid = 2
        cond = (hx >> 28) & 0xF
        R = (hx >> 22) & 0x1
        mask = (hx >> 16) & 0xF
        imm12 = (hx >> 0) & 0xFFF
        rethx = 0x0F000000
        rethx = rethx | (cond << 28)
        rethx = rethx | (inst_id << 20)
        rethx = rethx | (inst_subid << 17)
        rethx = rethx | (mask << 13)
        rethx = rethx | (imm12 << 1)
        rethx = rethx | (R << 0)
        return rethx
```

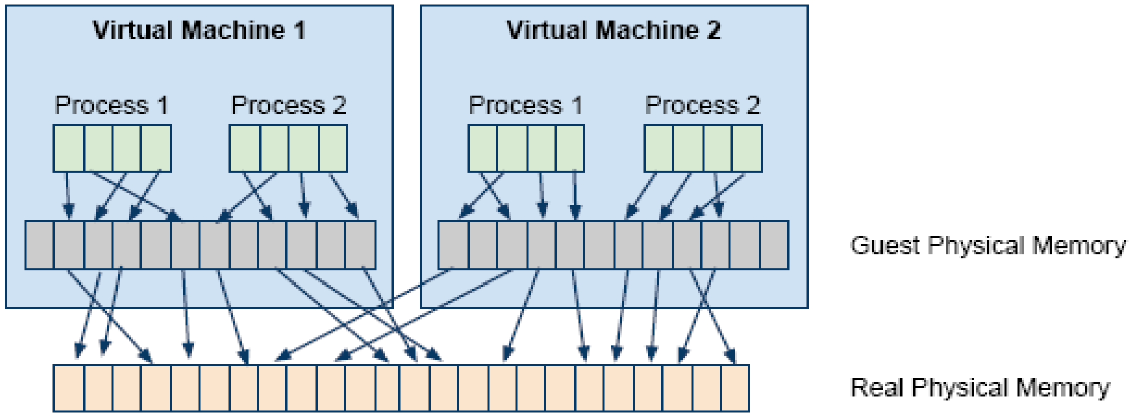
Xvisor utilizes cpatch to convert all problematic instructions for OS image files (ELF format).

Requirements of real Hypervisor

- VMM at higher privilege level than VMs
  - CPU Virtualization
  - Memory Virtualization
  - Device & I/O Virtualization
- User and System modes
- Privileged instructions only available in system mode
  - Trap to system if executed in user mode
- All physical resources only accessible using privileged instructions
  - Including page tables, interrupt controls, I/O registers

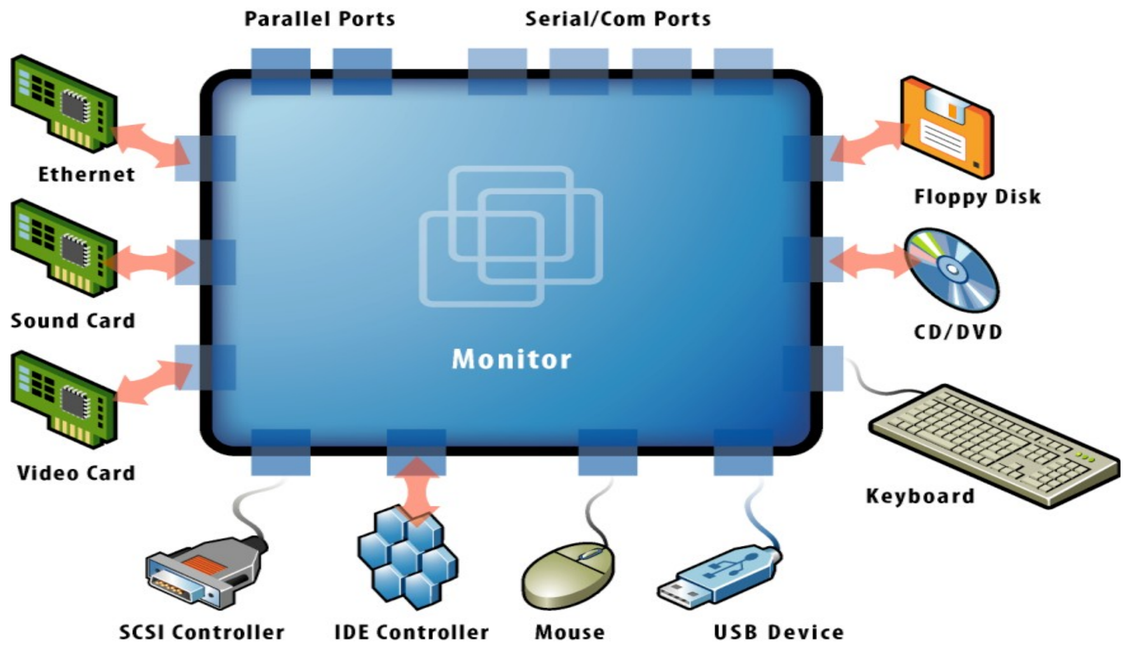
Memory Virtualization

- Deal with allocation of Physical memory among Guest OS
- RAM space shares among Guest OS
- Processors with memory virtualization support is expecting in 2nd generation processors (Intel VT and ARM Cortex-A15)



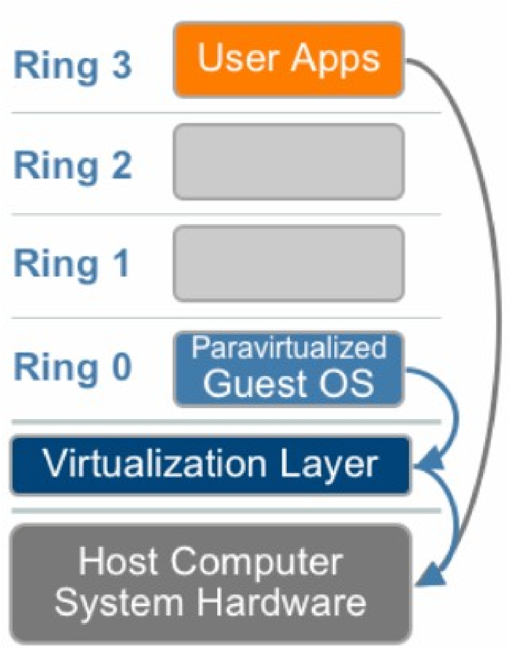
Device and I/O Virtualization

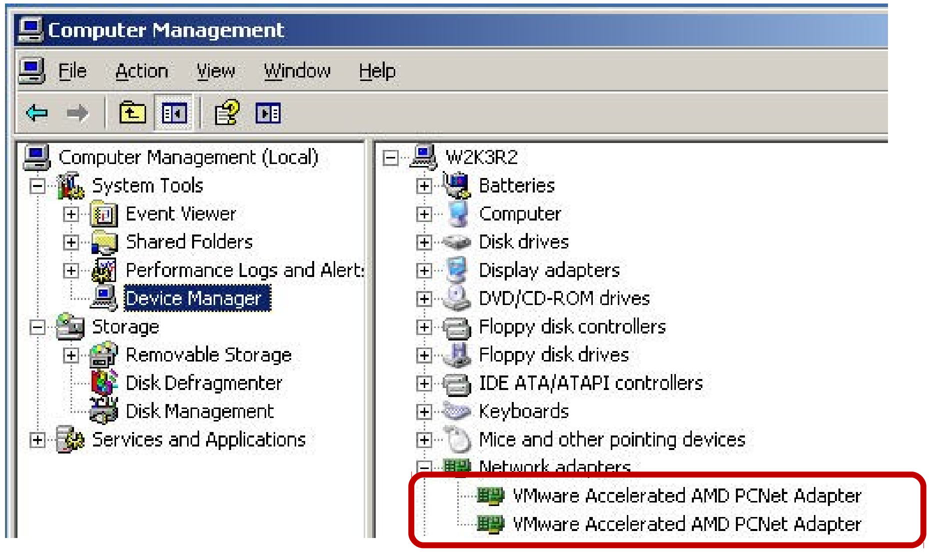
- Deal with routing of I/O requests between virtual devices and the shared physical hardware
- Similar to the single I/O device shared concurrently among different applications.
- Hypervisor virtualizes the physical hardware and present each virtual machine with a standard set of virtual devices



Paravirtualization

- OS or system devices are virtualization aware
- Requirements:
  - OS level – translated/modified kernel
  - Device level – paravirtualized or “enlightened” device drivers





Paravirtualization

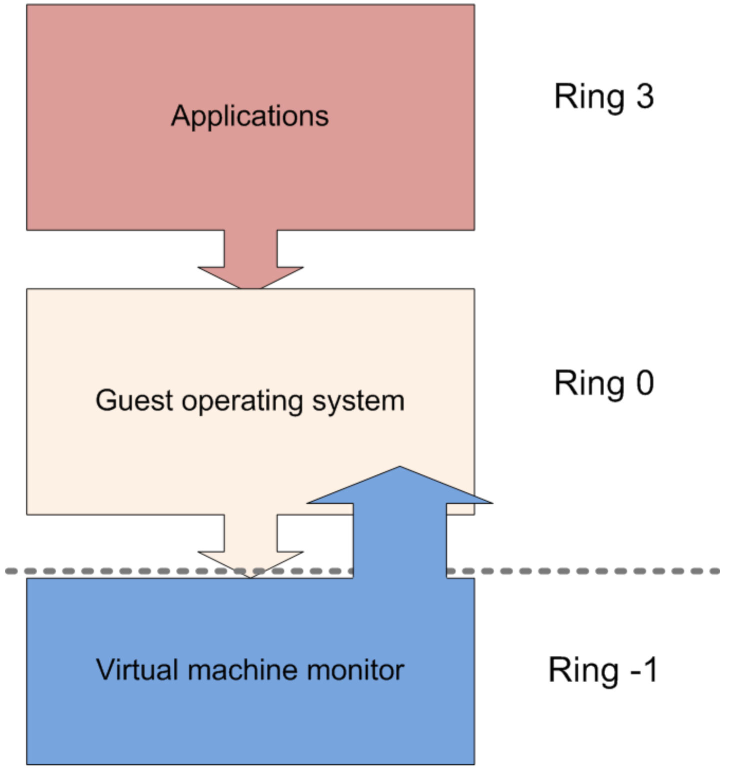
- Why all the trouble? Just “port” a guest operating system to the interface of your choice.
- Paravirtualization can
  - provide better performance
  - simplify VMM
- but at a maintainance cost and you need the source code
  - Compromise: Use paravirtualized drivers for I/O performance (KVM virtio, VMware).
- Examples: MkLinux, L4Linux, Xen, . . .

Paravirtualization

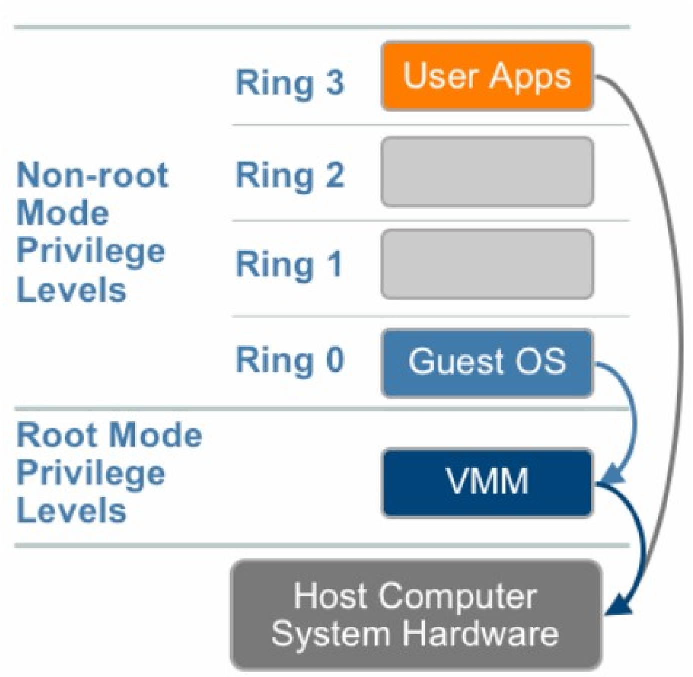
- Paravirtualization can also be semi-automated:
  - Sensitive instructions are automatically identified (in compiler output).
  - Sensitive memory access needs to manually identified.
  - Leave markers in binary.
  - On VM load-time, VMM replaces instructions with emulation code.
  - In-Place VMM translates to hypervisor calls.
- Benefits:
  - less effort than plain paravirtualization
  - comparable speed

Hardware-assisted Virtualization

- Hardware is virtualization aware
- Hypervisor and VMM load at Ring -1
- Remove CPU emulation bottleneck
- Provides address bus isolation



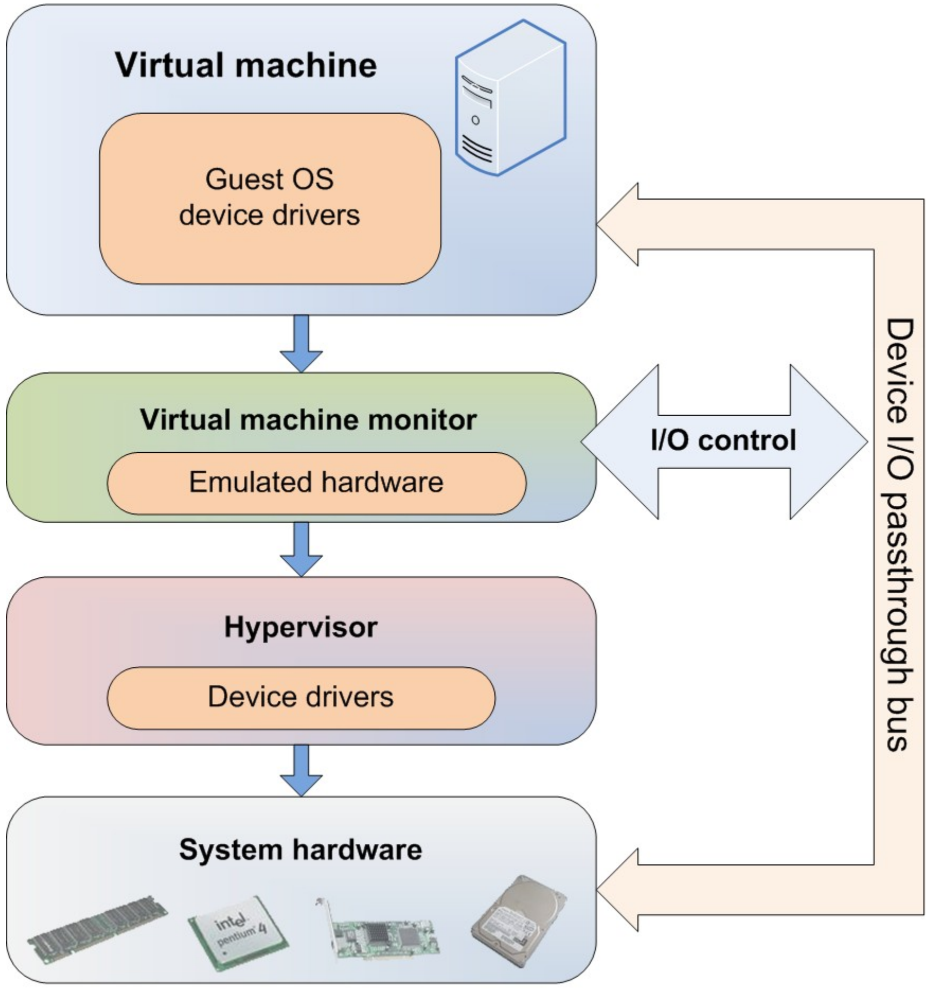
Hardware-assisted virtualization


Ring 3
User Apps
Non-root Mode Privilege Levels
Ring 2
Ring 1
Ring 0
Guest OS
Root Mode Privilege Levels
VMM
Host Computer System Hardware

Hardware-assisted Virtualization

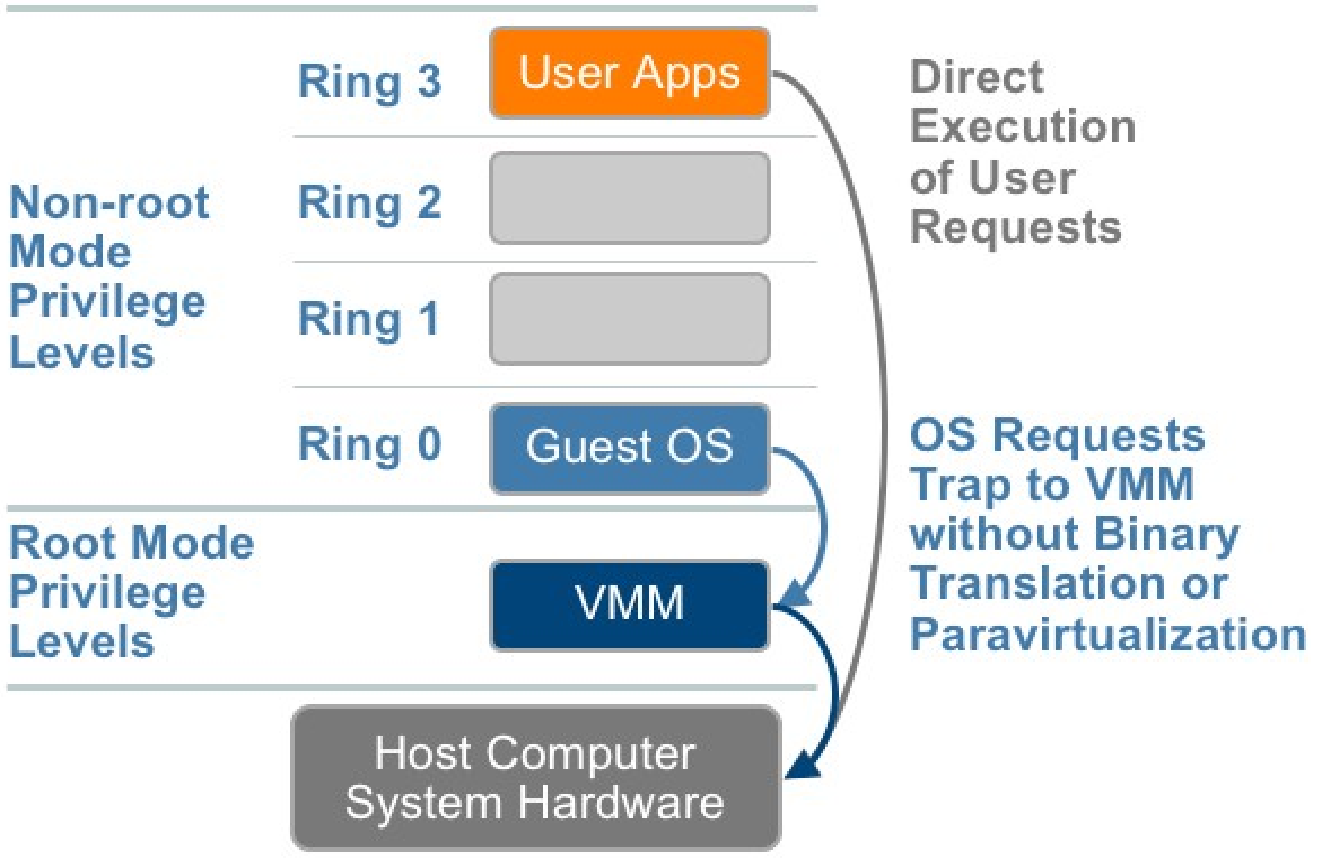
- VMM coordinates direct hardware access
- Memory virtualization solved in 2nd generation hardware assisted platforms
- Passthrough I/O has limited use cases without IOV (I/O Virtualization)

  http://www.pcisig.com/specifications/iov/



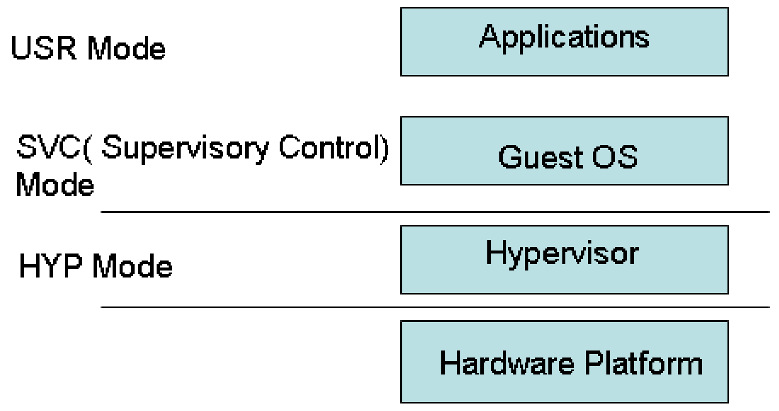
Hardware-assisted Virtualization in x86

- VT technology enables new execution mode (VMX-Root Mode in x86 by Intel) in the processors to support virtualization
- Hypervisor runs in a root mode below Ring0
- OS requests trap VMM without binary translation or PV
- Specialized Hardware support is required
- A special CPU privileged mode is to be selected to support


Ring 3
User Apps
Direct Execution of User Requests
Non-root Mode Privilege Levels
Ring 2
Ring 1
Ring 0
Guest OS
OS Requests Trap to VMM without Binary Translation or Paravirtualization
Root Mode Privilege Levels
VMM
Host Computer System Hardware

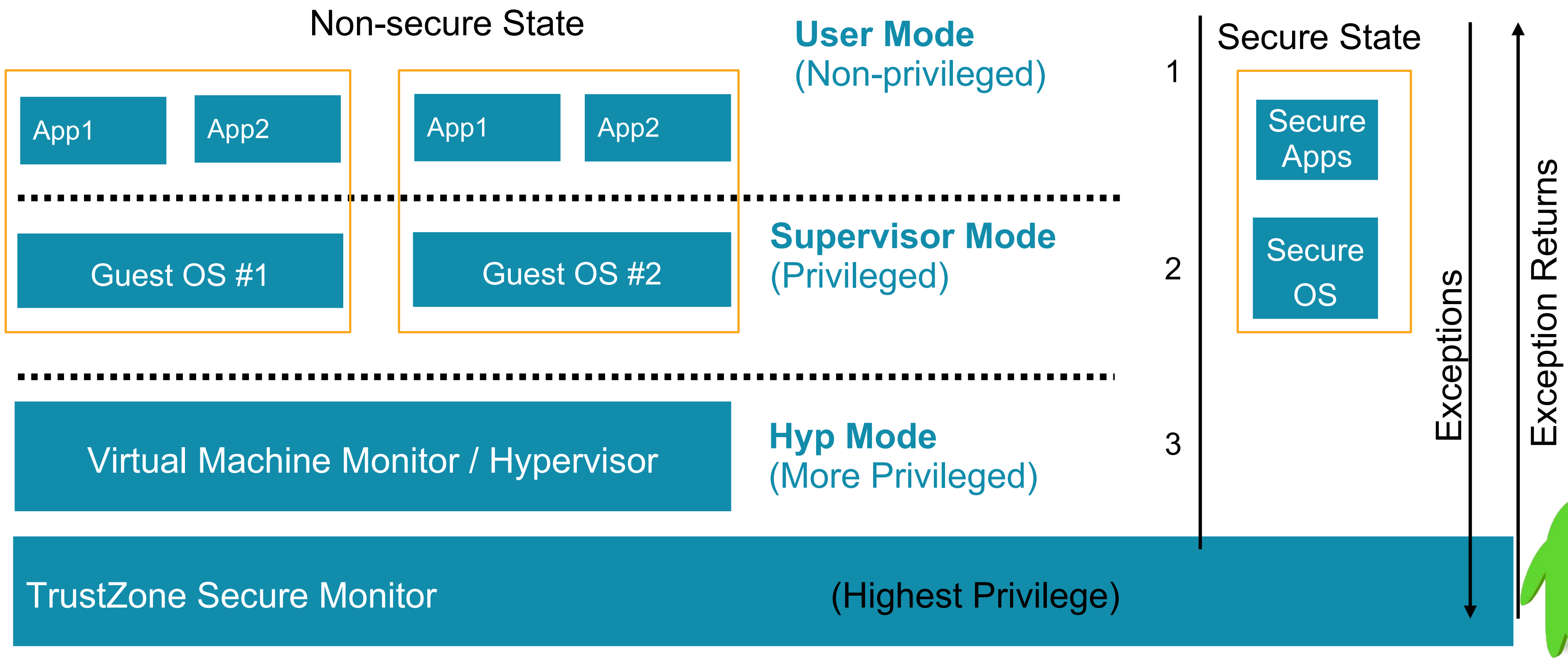
Hardware-assisted Virtualization in ARM

- Enable new execution mode Hypervisor (HYP)
- Hypervisor runs in a Hypervisor (HYP) mode
- Guest OS Runs in Supervisory Control (SVC) mode
- Applications runs in User (USR) mode



Virtualization: Third Privilege

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

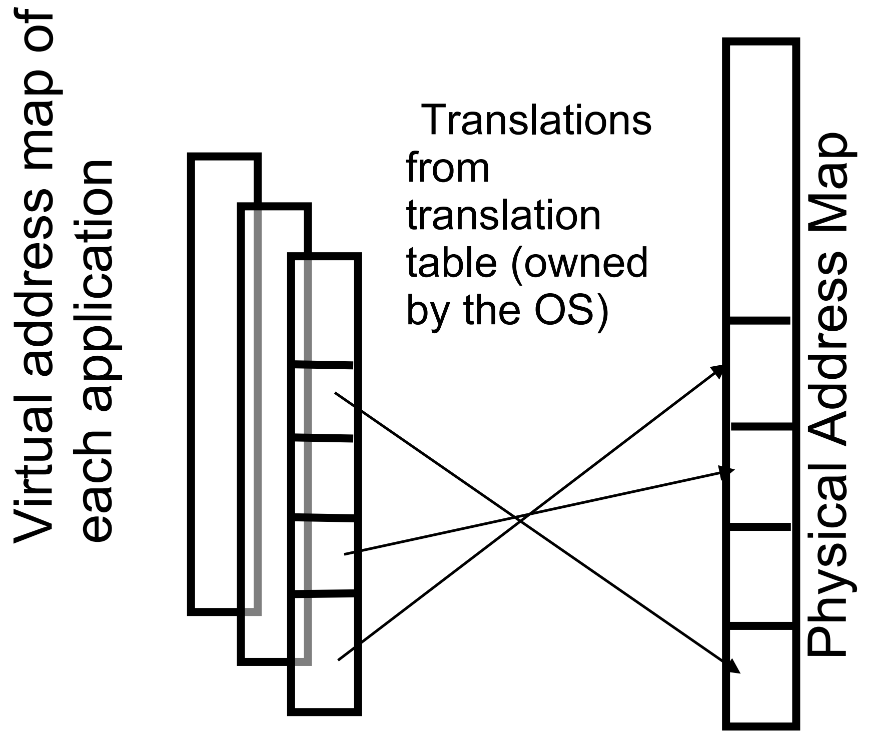


Virtualization Extensions: The Basics

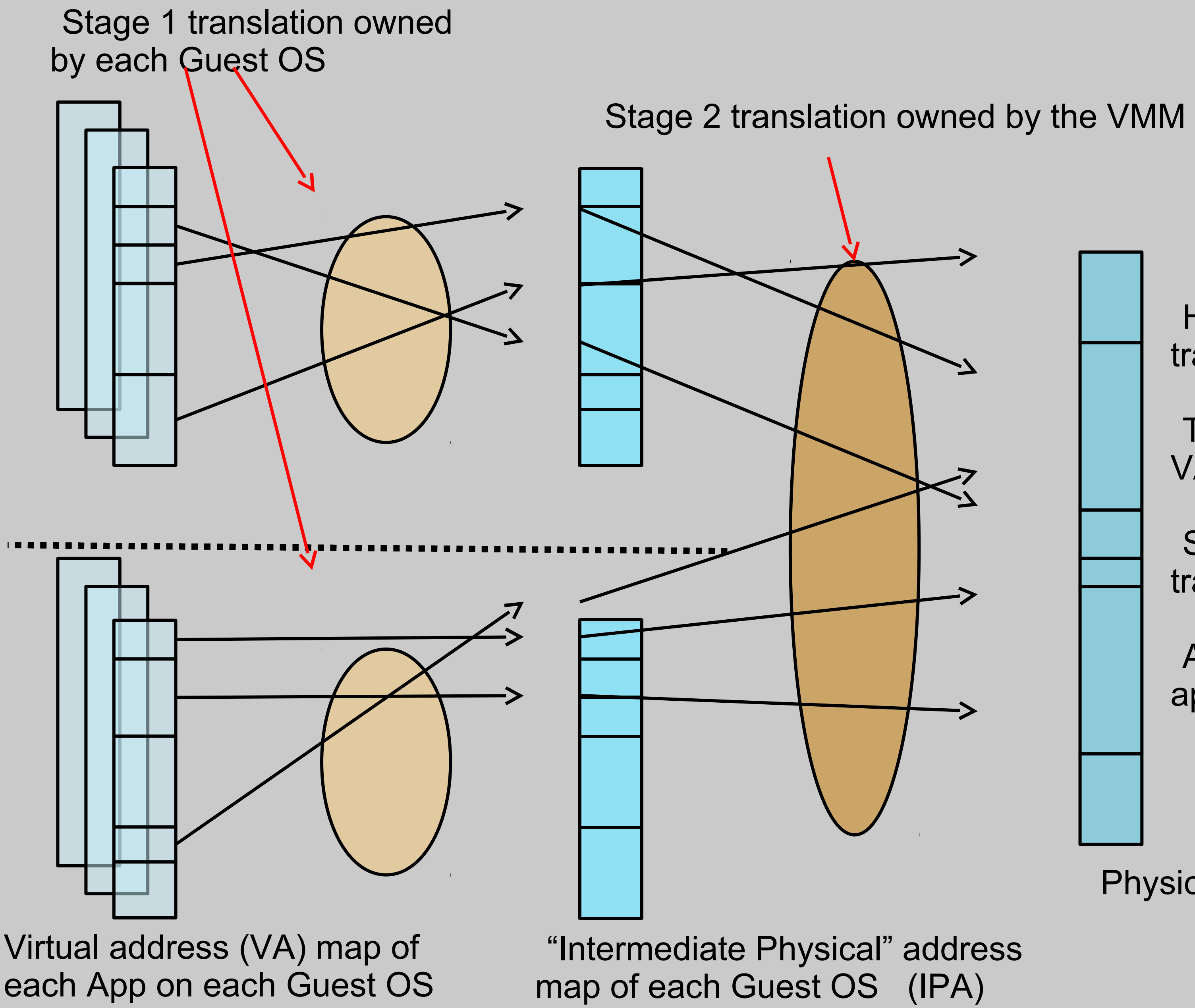
- New Non-secure level of privilege to hold Hypervisor
  - Hyp mode
- New mechanisms avoid the need Hypervisor intervention for:
  - Guest OS Interrupt masking bits
  - Guest OS page table management
  - Guest OS Device Drivers due to Hypervisor memory relocation
  - Guest OS communication with the interrupt controller (GIC)
- New traps into Hyp mode for:
  - ID register accesses and idling (WFI/WFE)
  - Miscellaneous “difficult” System Control Register cases
- New mechanisms to improve:
  - Guest OS Load/Store emulation by the Hypervisor
  - Emulation of trapped instructions through syndromes

Memory - the Classic Resource

- Before virtualization: the OS owns the memory
  - Allocates areas of memory to the different applications
  - Virtual Memory commonly used in “rich” operating systems


Virtual address map of each application
Translations from translation table (owned by the OS)
Physical Address Map

Virtual Memory in Two Stages


Stage 1 translation owned by each Guest OS
Stage 2 translation owned by the VMM
Virtual address (VA) map of each App on each Guest OS
"Intermediate Physical" address map of each Guest OS (IPA)
Physical Address (PA) Map

Hardware has 2-stage memory translation

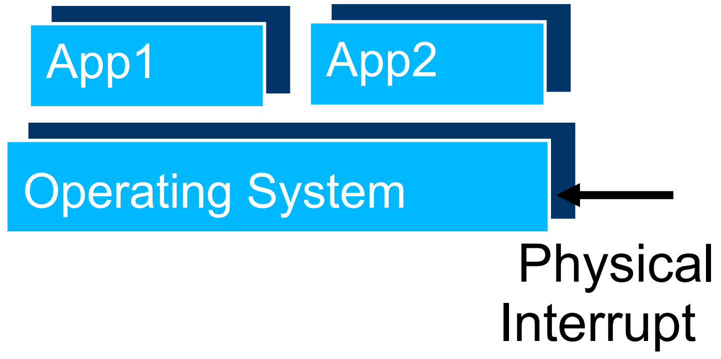
Tables from Guest OS translate VA to IPA

Second set of tables from VMM translate IPA to PA

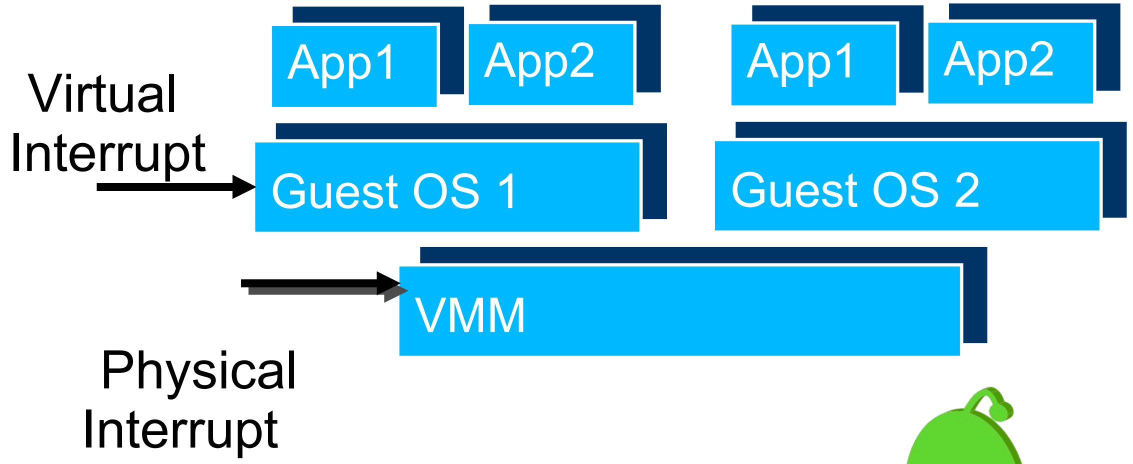
Allows aborts to be routed to appropriate software layer

Classic Issue: Interrupts

- An Interrupt might need to be routed to one of
  - Current or different Guest OS
  - Hypervisor
  - OS/RTOS running in the secure TrustZone environment
- Basic model of the ARM virtualization extensions
  - Physical interrupts are taken initially in the Hypervisor
  - If the Interrupt should go to a Guest OS, Hypervisor maps a "virtual" interrupt for that Guest OS



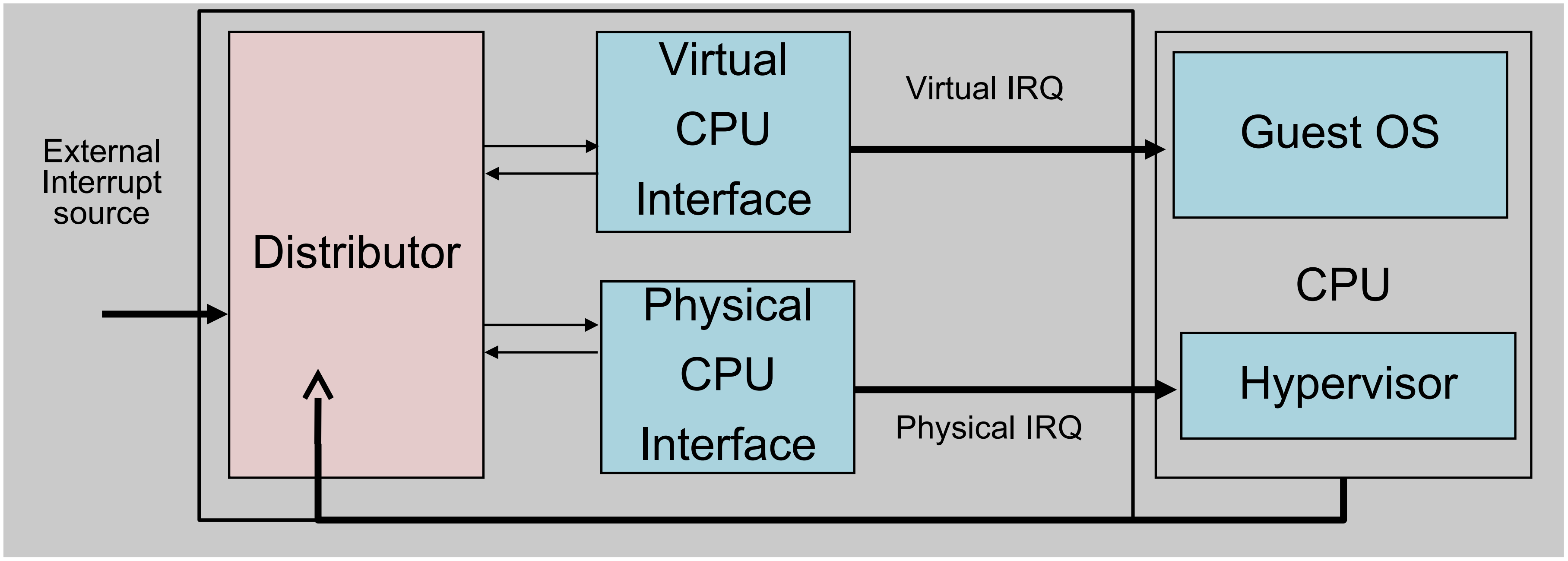
System without virtualization



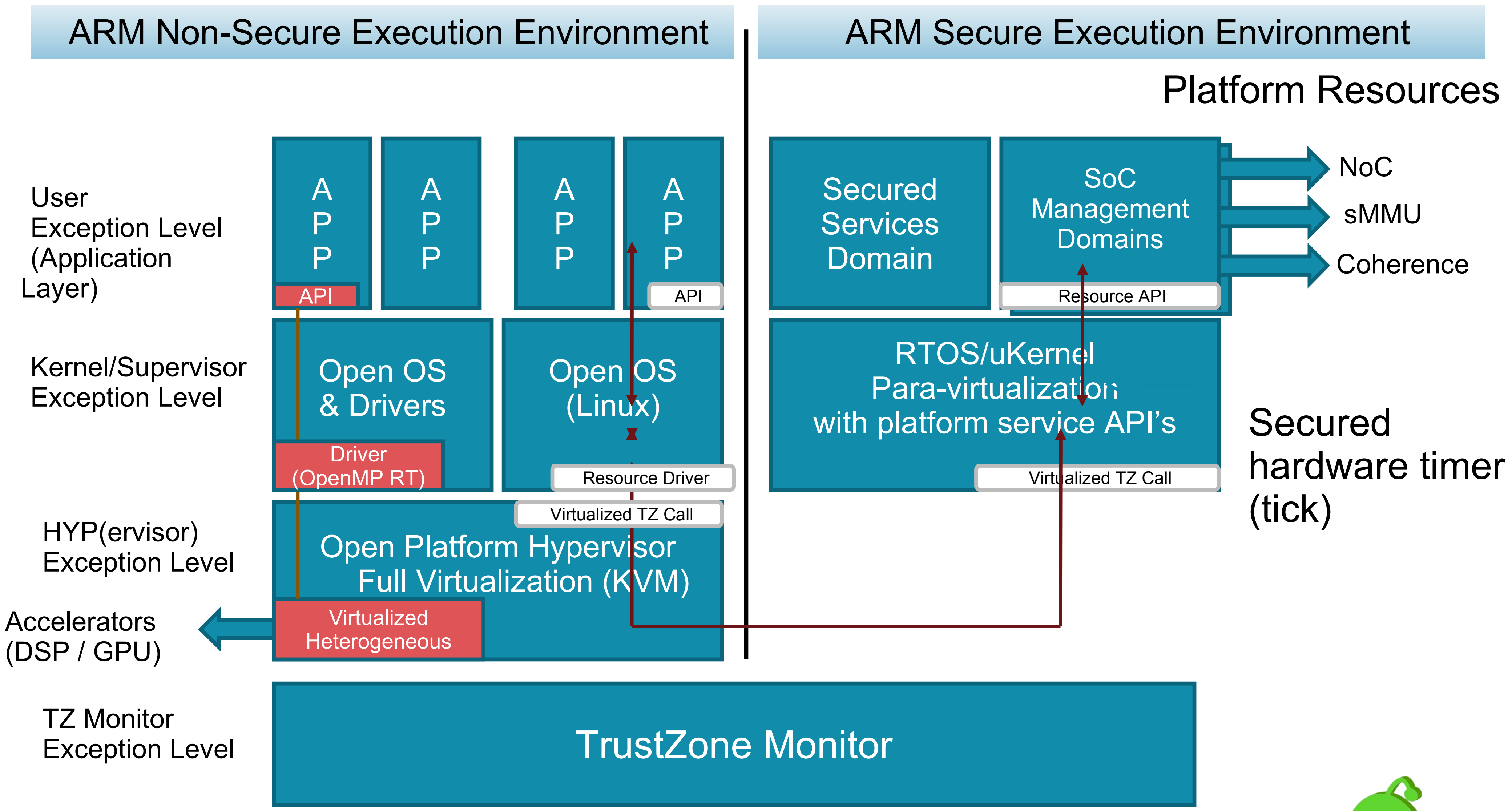
System with virtualization

Virtual interrupt example

- 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



Hardware based Hypervisor Framework


ARM Non-Secure Execution Environment
ARM Secure Execution Environment
Platform Resources
User Exception Level (Application Layer)
Kernel/Supervisor Exception Level
HYP(ervisor) Exception Level
Accelerators (DSP / GPU)
TZ Monitor Exception Level
APP
APP
APP
APP
API
API
Secured Services Domain
SoC Management Domains
Resource API
NoC
sMMU
Coherence
Open OS & Drivers
Driver (OpenMP RT)
Open OS (Linux)
Resource Driver
RTOS/uKernel Para-virtualization with platform service API's
Virtualized TZ Call
Virtualized TZ Call
Open Platform Hypervisor Full Virtualization (KVM)
Virtualized Heterogeneous
Secured hardware timer (tick)
TrustZone Monitor

Source: Hardware accelerated Virtualization in the ARM Cortex™ Processors, John Goodacre, ARM Ltd. (2011)

Embedded Virtualization Implementations

Embedded Hypervisors for ARM

(open source part)

- Xen
  - Xen-arm, contributed by **Samsung**

    ARM9, ARM11, ARM Cortex-A9 MP
  - Xen-arm-cortext-a15, contributed by **Citrix** - https://lkml.org/lkml/2011/11/29/265

    ARM Cortex-A15
- OKL4 (from open to close source), OKLabs
- L4Linux, TU Desden
- ARM Virtualizer (big.LITTLE switcher), Linaro
- KVM ARM
  - Columbia University, Linaro
  - NTHU, Taiwan
- Xvisor: supports ARMv5, ARMv7, ARMv7+VE
- Codezero

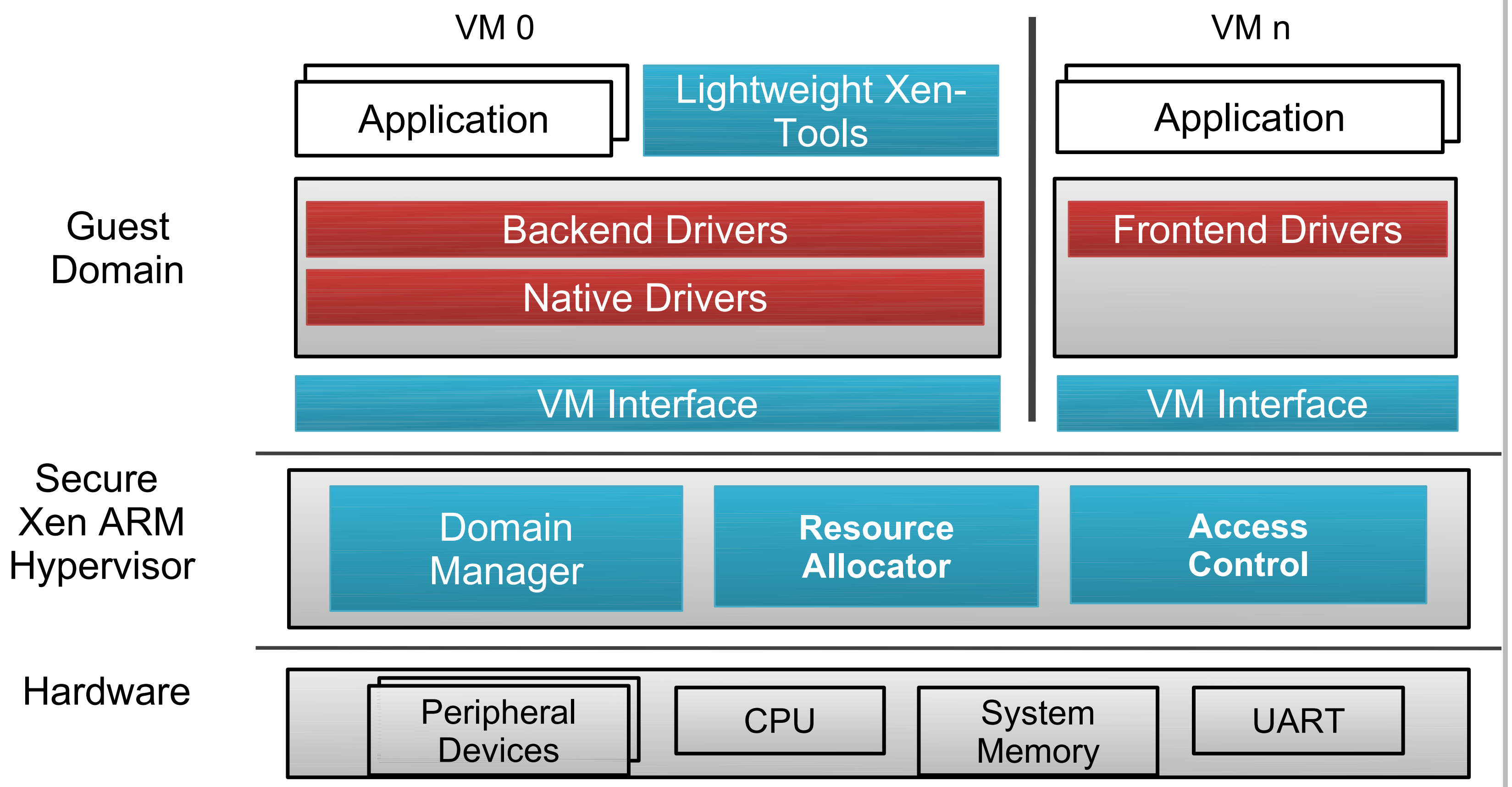
Xen-ARM (Samsung)

Goals

Lightweight virtualization for secure 3G/4G mobile devices

- High performance hypervisor based on ARM processor
- Fine-grained access control fitted to mobile devices

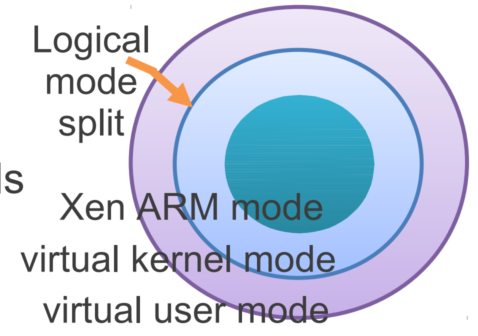
Architecture of Xen ARM



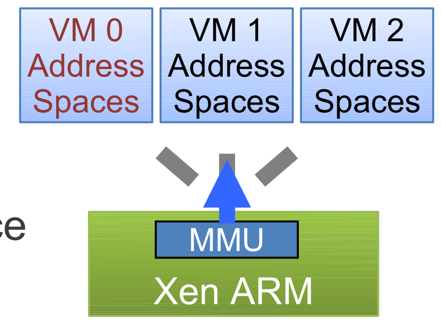
Xen-ARM (Samsung)

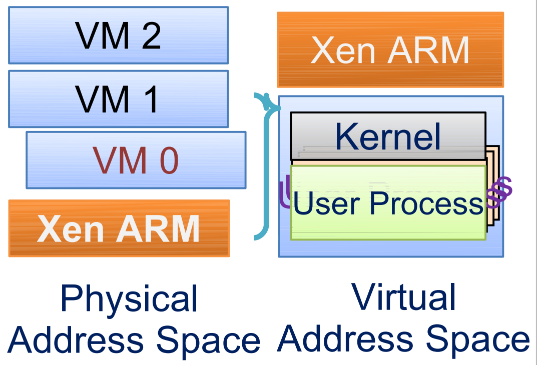
Overview

- CPU virtualization
- Virtualization requires 3 privilege CPU levels, but ARM supports 2 levels
  - Xen ARM mode: supervisor mode (most privileged level)
  - Virtual kernel mode: User mode (least privileged level)
  - Virtual user mode: User mode (least privileged level)

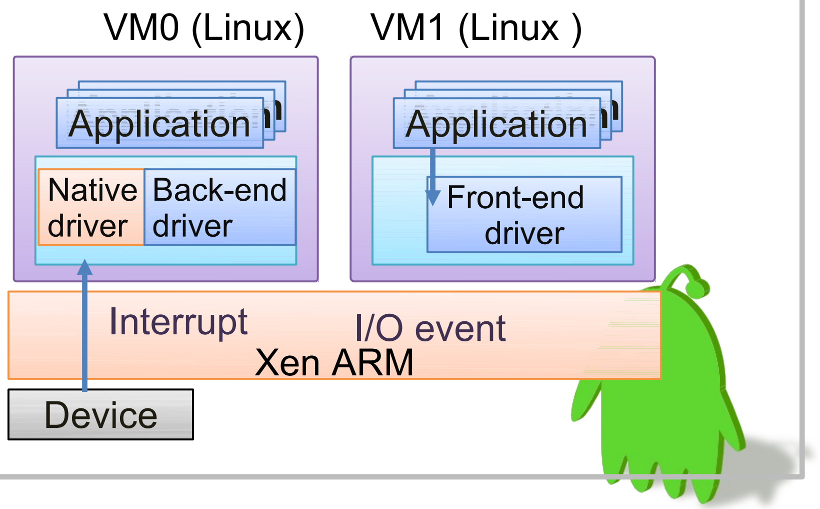


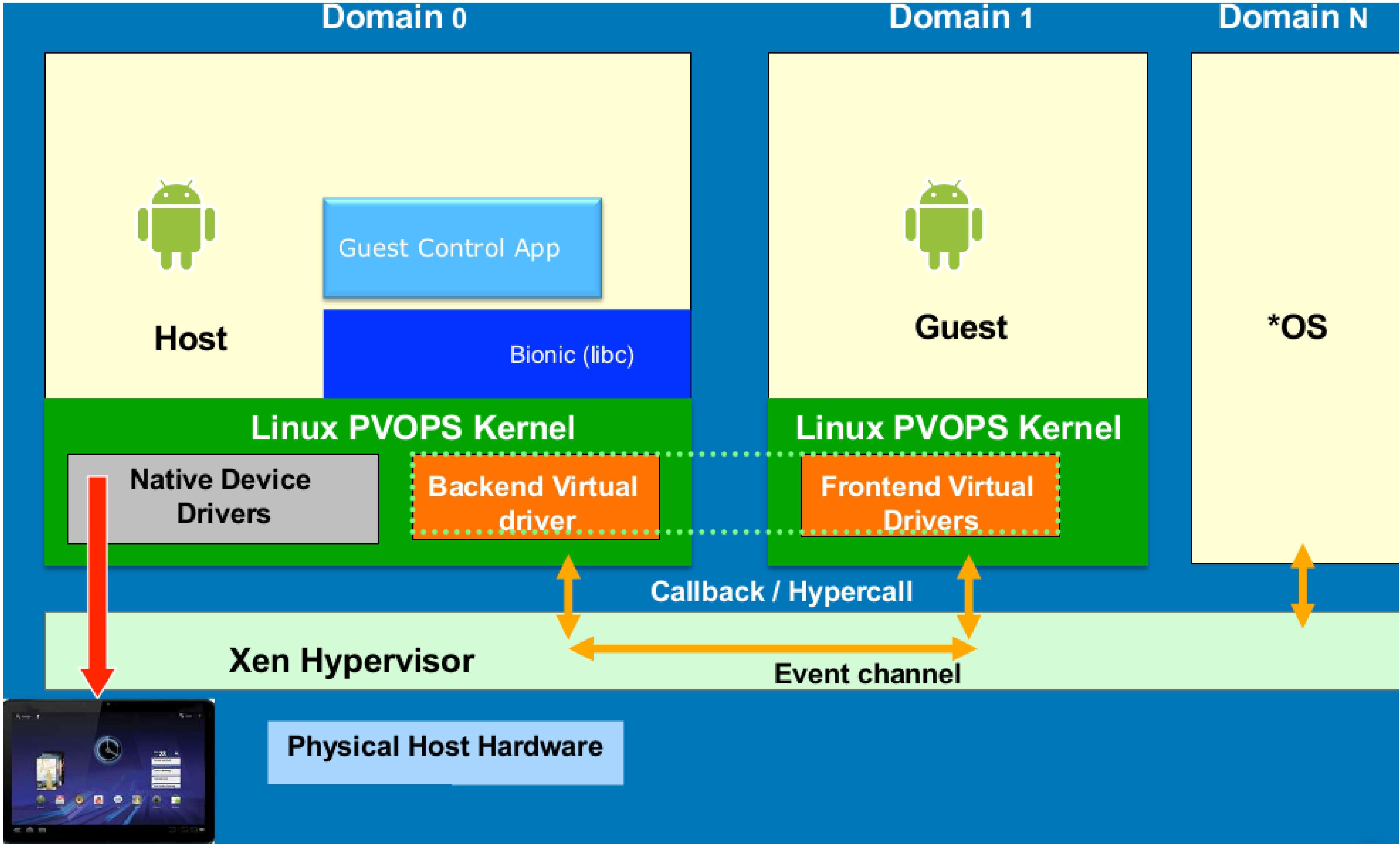
- Memory virtualization
- VM's local memory should be
- protected from other VMs
- Xen ARM switches VM's virtual address space
  - using MMU
  - VM is not allowed to manipulate MMU directly



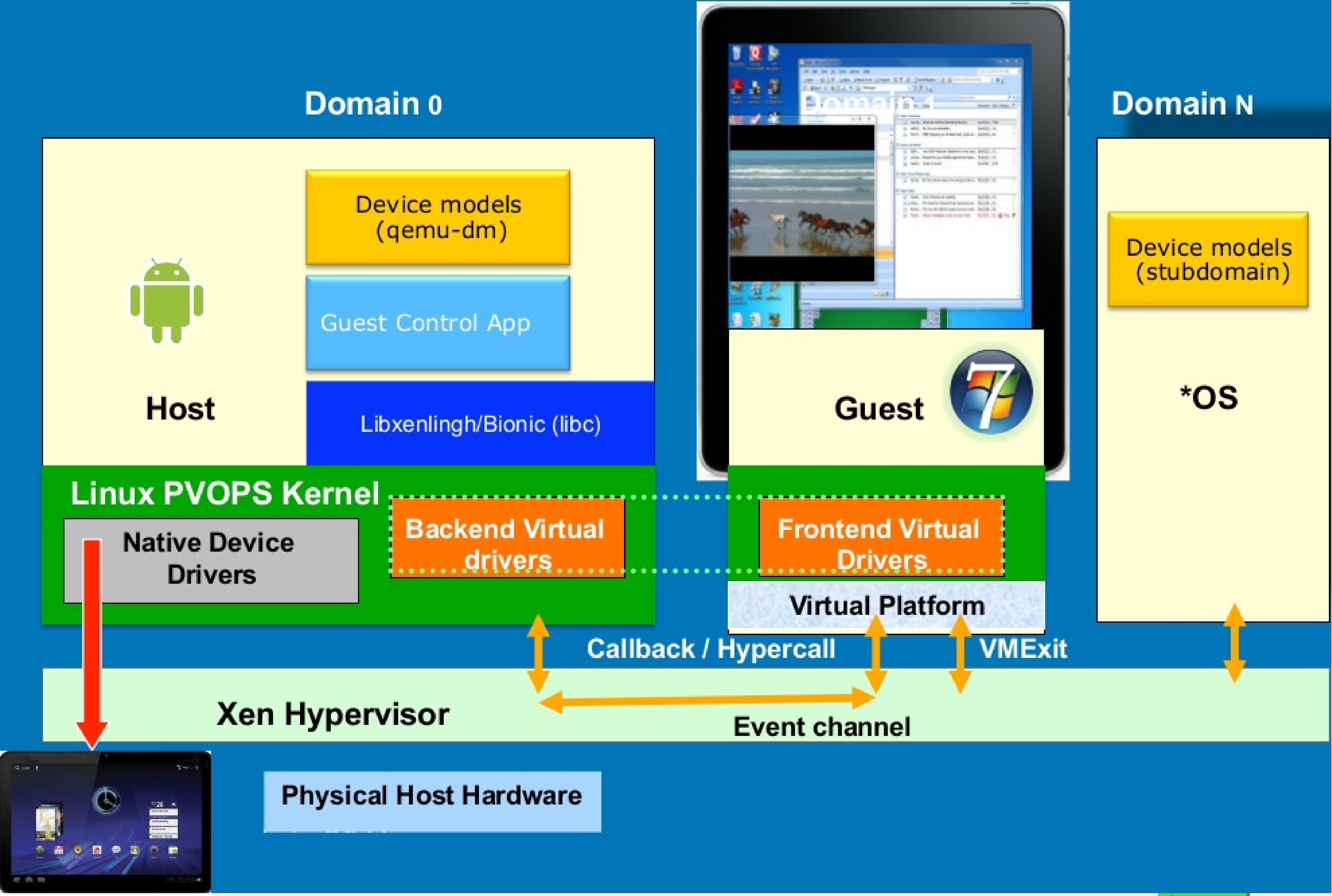


- I/O virtualization
- Split driver model of Xen ARM
  - Client & Server architecture for shared I/O devices
    - Client: frontend driver
    - Server: native/backend driver





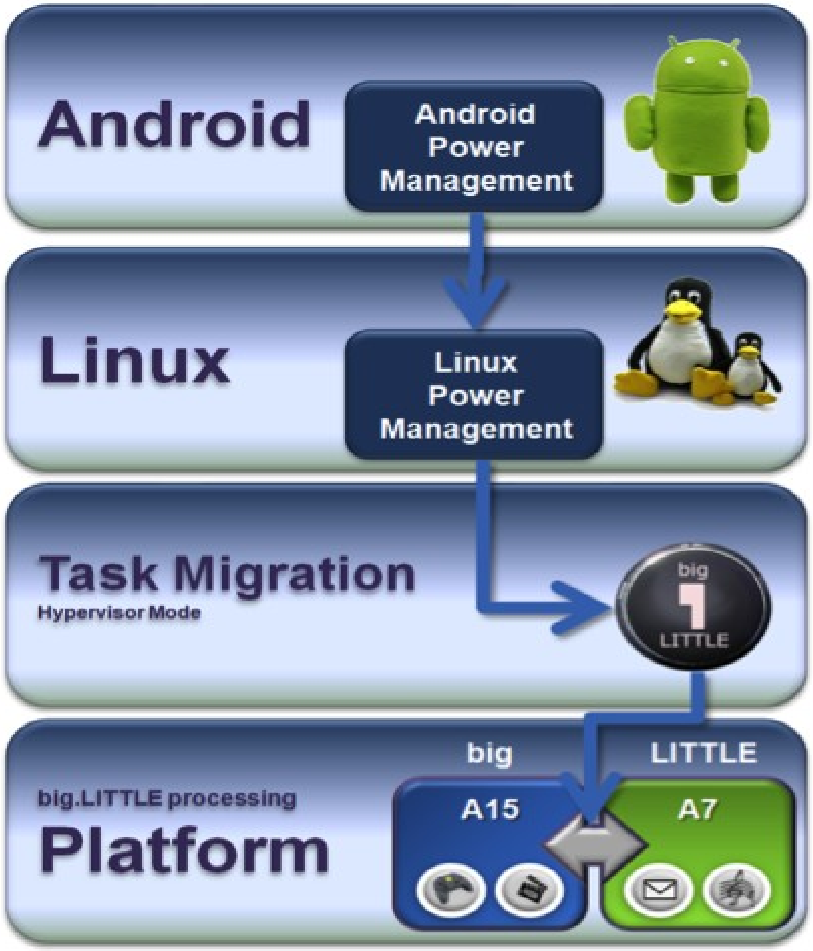
- Xen without assisted hardware VM



- Xen with assisted hardware VM

ARM Virtualizer for big.LITTLE

- big.LITTLE Task Migration model
  - software can seamlessly migrate from one processor to the other, depending on the use context and resulting performance requirements.
  - To lower even further the power, each core has its own Level 2 cache memory.
  - While sharing an L2 cache would be a more area optimized design, integration of an L2 cache on each processor, yields better power results.
- Achieved by having the software run not on the hardware, but on-top of a new layer operating in HYP mode and performing the task-migration.

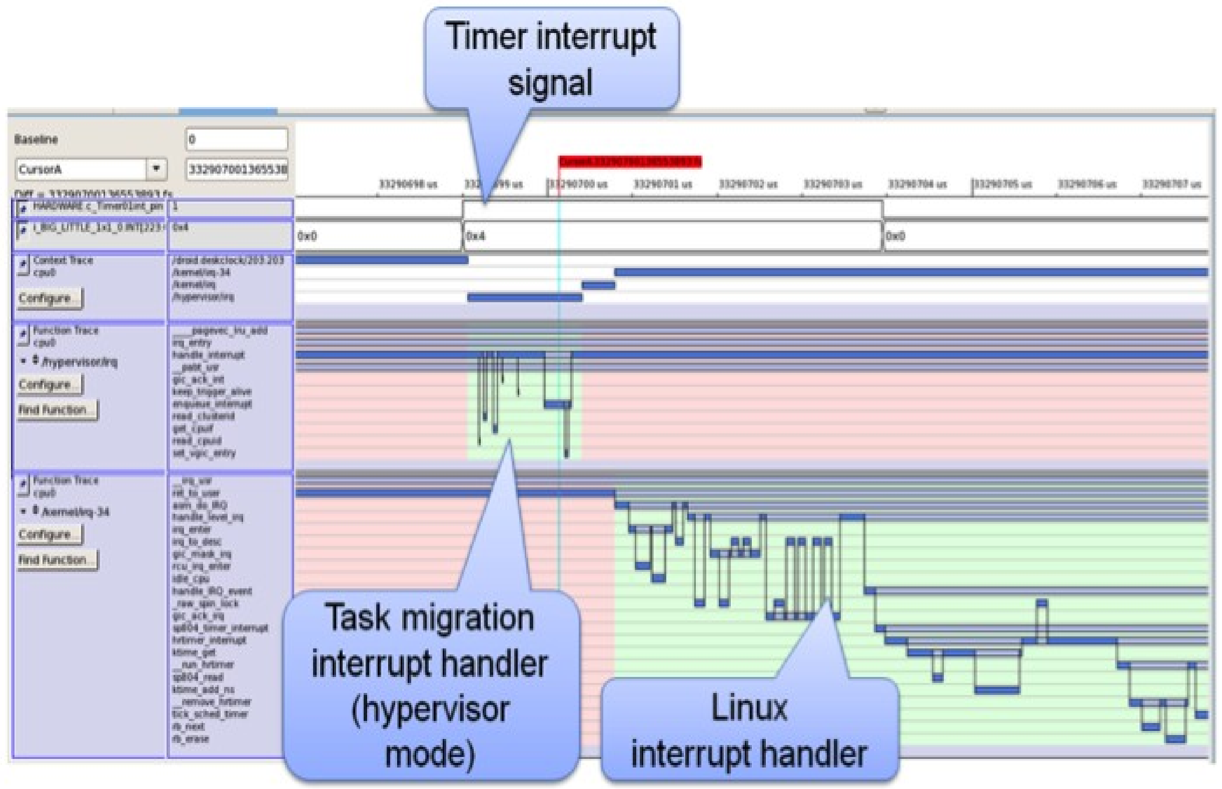


big.LITTLE: Hypervisors and Interrupts

- Why is hypervisor needed? Interrupts as an example
  - When we run multiple guest OSes on a system, interrupts coming from the hardware could be for either of the OSes.
  - A hypervisor needs to first intercept the interrupt from the system, and then decide to which guest OS it was addressed.
- For big.LITTLE processing, multiple processor clusters sharing the same interrupt controller. The hypervisor ensures the transparency of the clusters for the OS and does the task migration.
  - hypervisor needs its own interrupts and should not interfere with the OS.

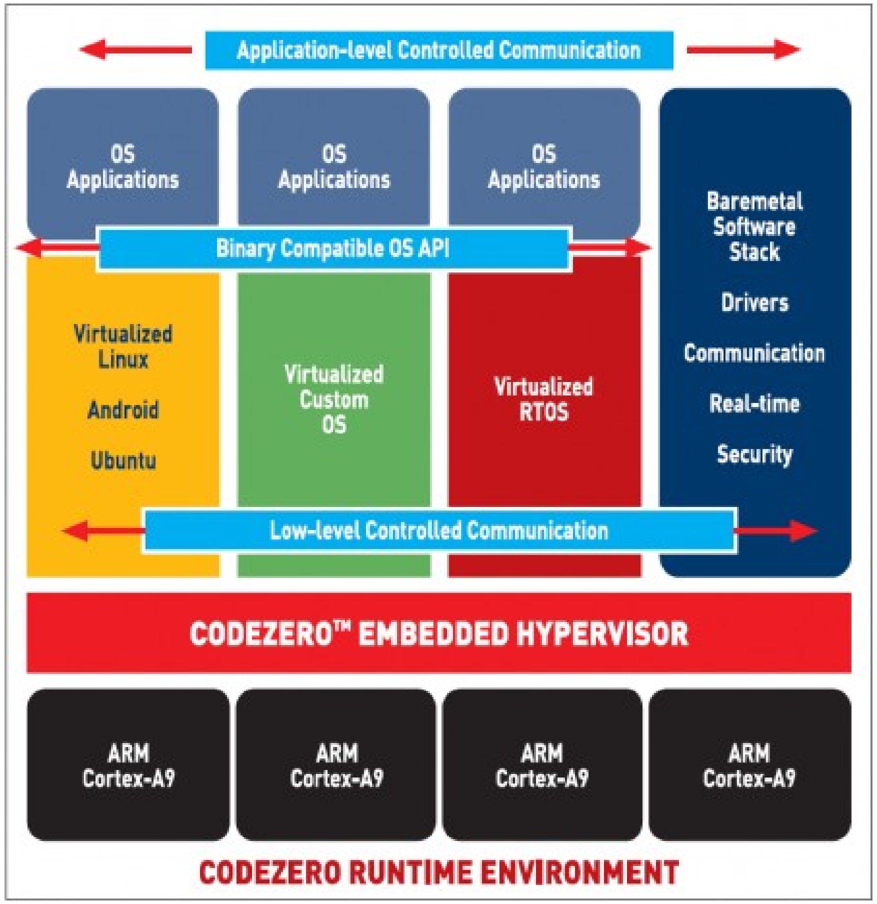
Hardware Supported Interrupts Virtualization

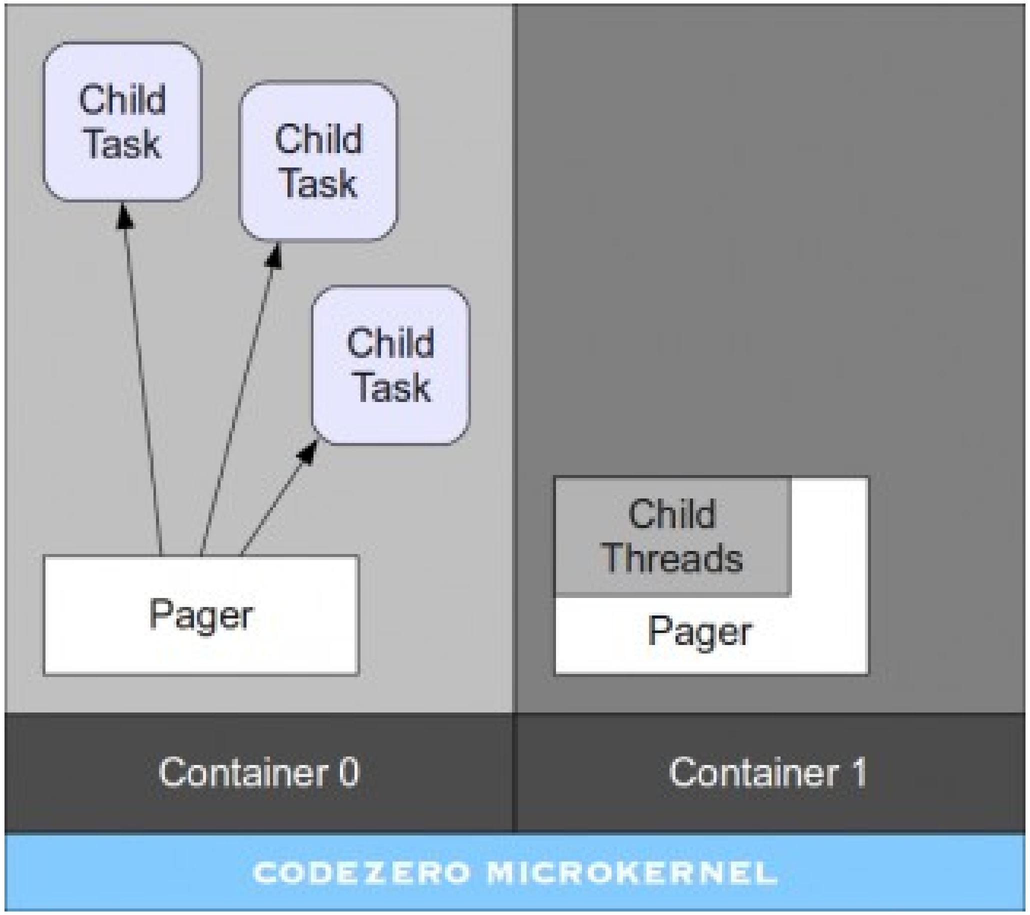
- In HYP mode, a higher privileged exception vector allows trapping interrupts even before the OS can react
- when an interrupt come in, hypervisor handles first.
  - If it is for OS, then it will configure a virtualized interrupt controller
  - OS will then handle the interrupt as if there was no hypervisor in between.

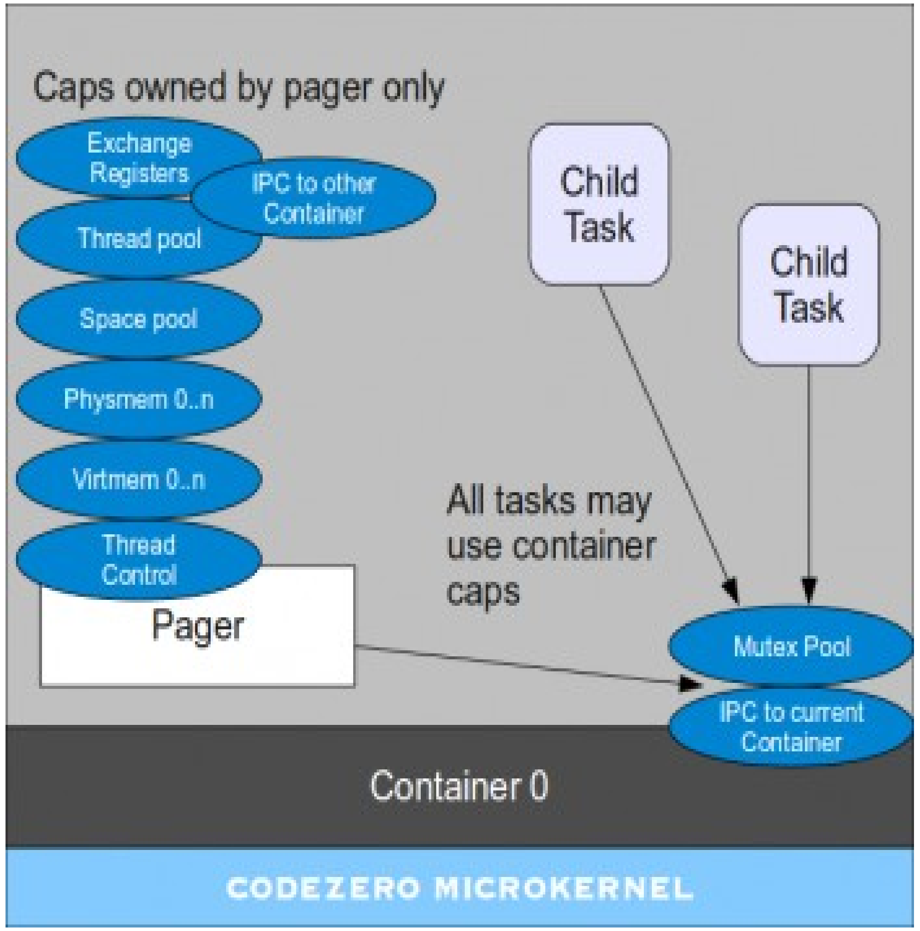


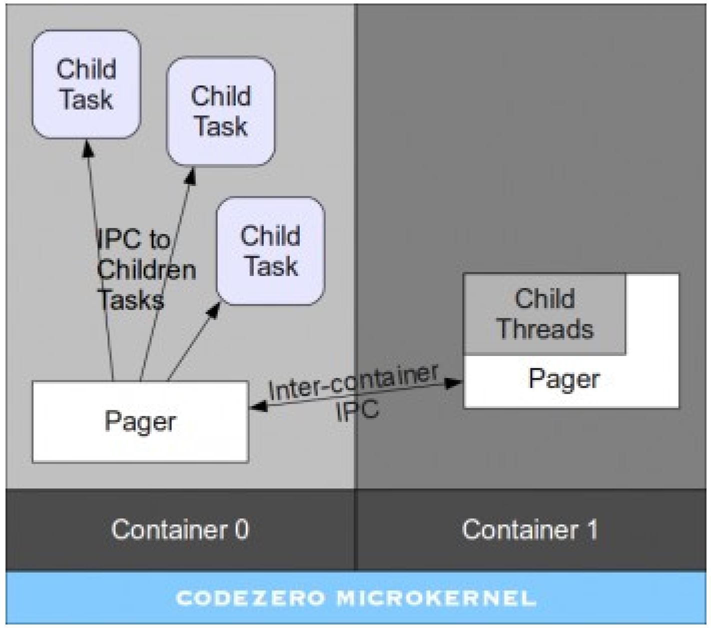
Codezero hypervisor

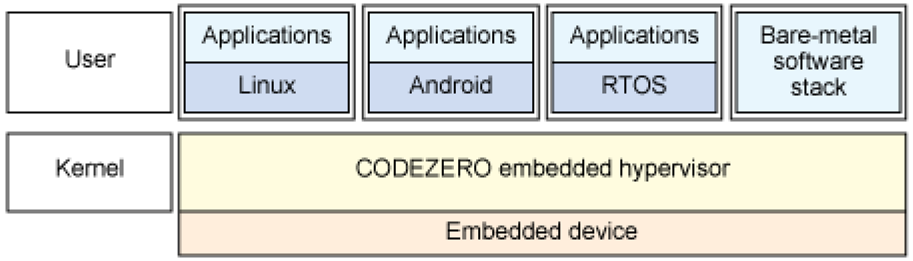
- Optimized for latest ARM cores (Cortex-A9/A15)
- L4 microkernel based design, written from scratch
- Capability based dynamic resource management
- Container oriented driver model: no modifications required for Linux




Child Task
Child Task
Child Task
Pager
Child Threads
Pager
Container 0
Container 1
CODEZERO MICROKERNEL


Caps owned by pager only
Exchange Registers
IPC to other Container
Thread pool
Space pool
Physmem 0..n
Virtmem 0..n
Thread Control
Pager
Child Task
Child Task
All tasks may use container caps
Mutex Pool
IPC to current Container
Container 0
CODEZERO MICROKERNEL


Child Task
Child Task
Child Task
IPC to Children Tasks
Pager
Inter-container IPC
Child Threads
Pager
Container 0
Container 1
CODEZERO MICROKERNEL

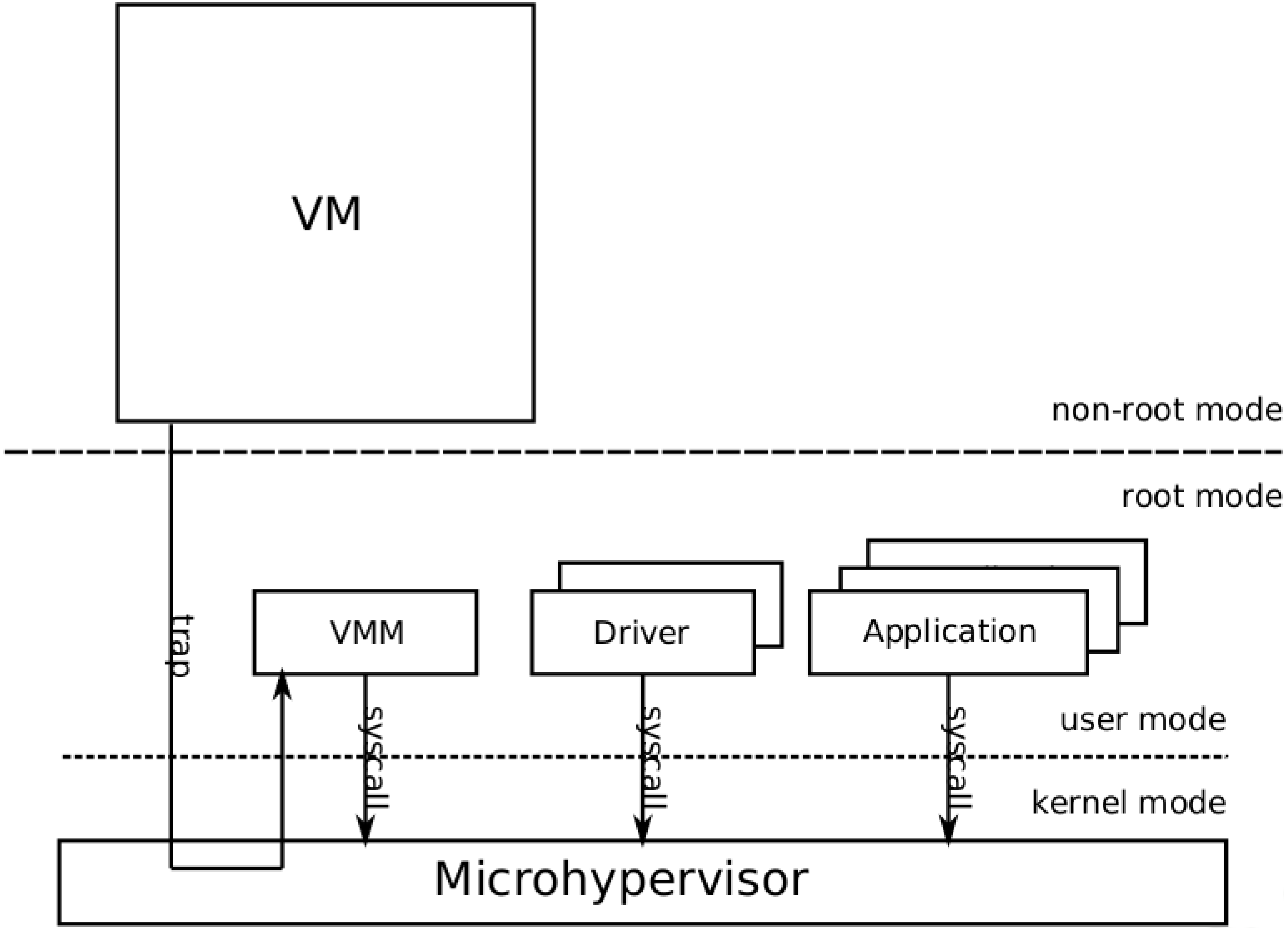

User
Applications
Linux
Applications
Android
Applications
RTOS
Bare-metal software stack
Kernel
CODEZERO embedded hypervisor
Embedded device

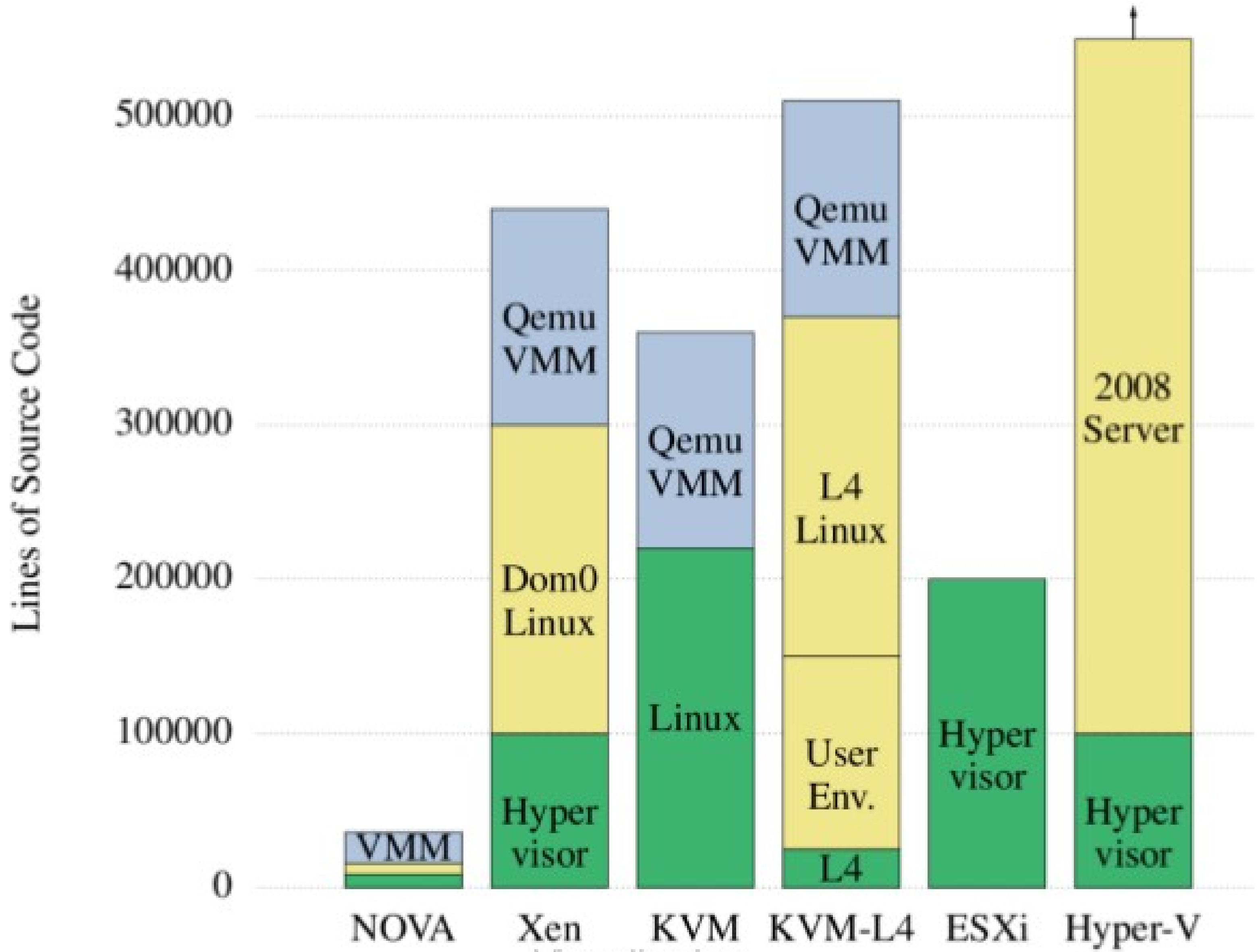
Micro-hypervisor

- Microvisor – OKL4 4.0
- Research projects such as NOVA, Coyotos, and seL4
- Aided by virtualizable ISA

- Microhypervisor
  - the “kernel” part
  - provides isolation
  - mechanisms, no policies
  - enables safe access to virtualization features to userspace

- VMM
  - the “userland” part
  - CPU emulation
  - device emulation


VM
non-root mode
root mode
trap
VMM
Driver
Application
syscall
syscall
syscall
user mode
kernel mode
Microhypervisor



Virtualization

Source: **VIRTUALIZATION**, Julian Stecklina, TU Dresden

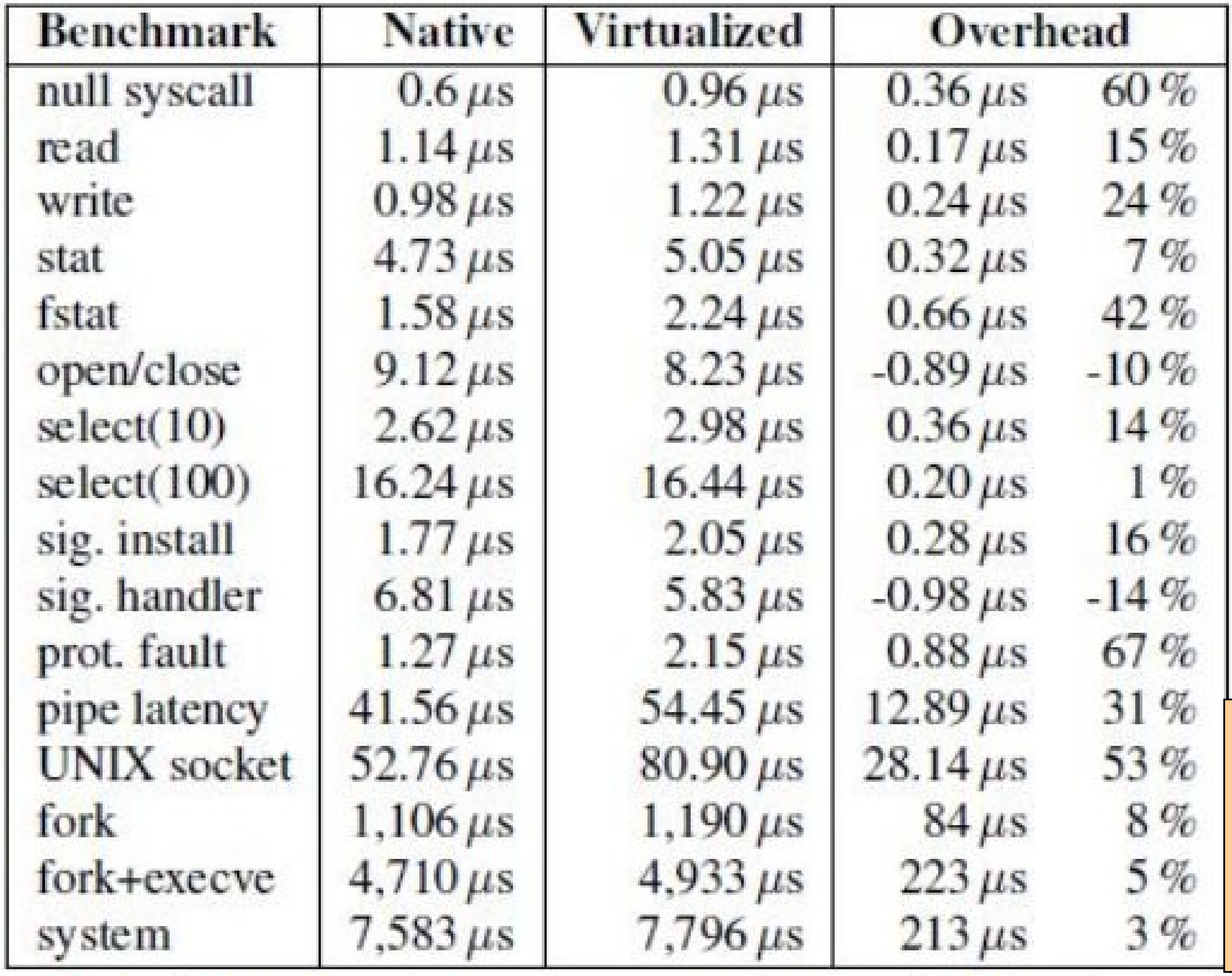
Advantage of NOA architecture: Reduce TCB of each VM

- Micro-hypervisor provides low-level protection domains
  - address spaces
  - virtual machines
- VM exits are relayed to VMM as IPC with selective guest state
- one VMM per guest in (root mode) userspace:
  - possibly specialized VMMs to reduce attack surface
  - only one generic VMM implemented

Guest OS specific issues

Known Issues when deploying Virtualization into Android based Devices

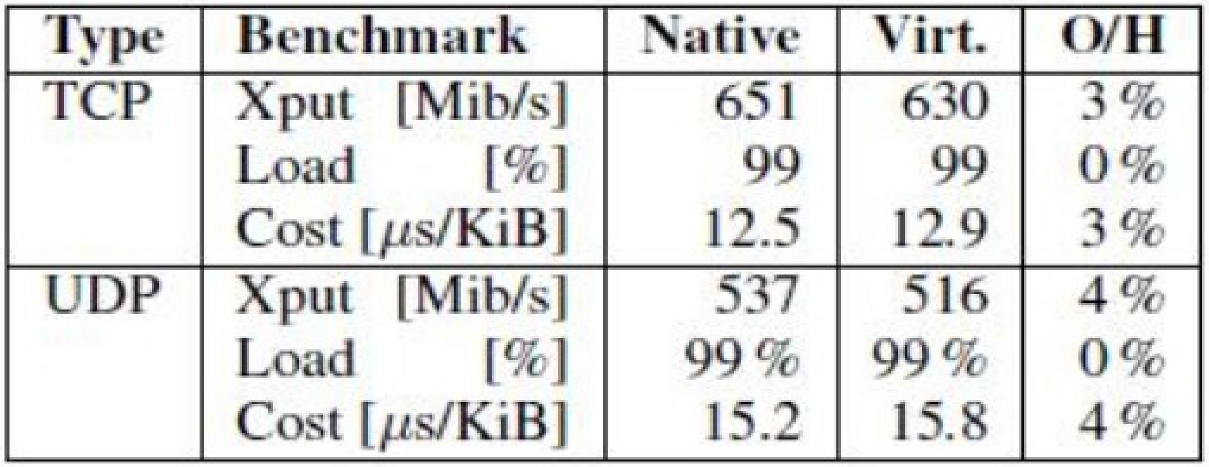
- Performance
  - system call, which needs a single hypercall to virtualize, is acceptable
  - Driver separation might be the problem: tradeoff
- Both Type I and Type II virtualization are deployed in real products
  - eventually becomes "hybrid" approaches.
  - being complex (best practice: GPU virtualization)
- LoC of Linux kernel modifications
- Power consumption
  - Enforced as critical resouce in mind
- Duplicated implementation in difference area



Benchmark	Native	Virtualized	Overhead	
null syscall	0.6 μs	0.96 μs	0.36 μs	60 %
read	1.14 μs	1.31 μs	0.17 μs	15 %
write	0.98 μs	1.22 μs	0.24 μs	24 %
stat	4.73 μs	5.05 μs	0.32 μs	7 %
fstat	1.58 μs	2.24 μs	0.66 μs	42 %
open/close	9.12 μs	8.23 μs	-0.89 μs	-10 %
select(10)	2.62 μs	2.98 μs	0.36 μs	14 %
select(100)	16.24 μs	16.44 μs	0.20 μs	1 %
sig. install	1.77 μs	2.05 μs	0.28 μs	16 %
sig. handler	6.81 μs	5.83 μs	-0.98 μs	-14 %
prot. fault	1.27 μs	2.15 μs	0.88 μs	67 %
pipe latency	41.56 μs	54.45 μs	12.89 μs	31 %
UNIX socket	52.76 μs	80.90 μs	28.14 μs	53 %
fork	1,106 μs	1,190 μs	84 μs	8 %
fork+execve	4,710 μs	4,933 μs	223 μs	5 %
system	7,583 μs	7,796 μs	213 μs	3 %

LmBench shows near native performance with OKL4 3.0 on ARMv7 target

Source: The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors, Gernot Heiser & Ben Leslie, Open Kernel Labs and NICTA (2010)



Type	Benchmark	Native	Virt.	O/H
TCP	Xput [Mib/s]	651	630	3 %
	Load [%]	99	99	0 %
	Cost [μs/KiB]	12.5	12.9	3 %
UDP	Xput [Mib/s]	537	516	4 %
	Load [%]	99 %	99 %	0 %
	Cost [μs/KiB]	15.2	15.8	4 %

NetPerf
fully-loaded CPU and the throughput degradation of the virtualized is only 3% and 4%.

Enhancements for Android virtualization

- Firmware OTA
- Policy based runtime security enhancemet
- Adaptive resource managemet
- Fast path IPC based on microkernel/hypervisor
- Faster device boot time for better user experience

Reference

- 前瞻資訊科技—虛擬化，薛智文，台大資訊所 (2011)
- Making Sense Of Virtualization, Achim Nohl, Synopsys
- ARM Virtualization: CPU & MMU Issues, Prashanth Bungale, vmware
- An Overview of Microkernel, Hypervisor and Microvisor Virtualization Approaches for Embedded Systems, Asif Iqbal, Nayeema Sadeque and Rafika Ida Mutia, Lund University, Sweden
- Virtualization for embedded systems, M. Tim Jones
- Hardware accelerated Virtualization in the ARM Cortex™ Processors, John Goodacre, ARM Ltd. (2011)
- Philippe Gerum, State of Real-Time Linux: Don't Stop Until History Follows, ELC Europe 2009

http://0xlab.org