Real-Time Virtualization

Real-Time Industrial Systems

Marcello Cinque

Roadmap

- Virtualization: overview
- Why virtualization in real-time systems?
- Real-Time hypervisors
  - RT-XEN
  - Jailhouse

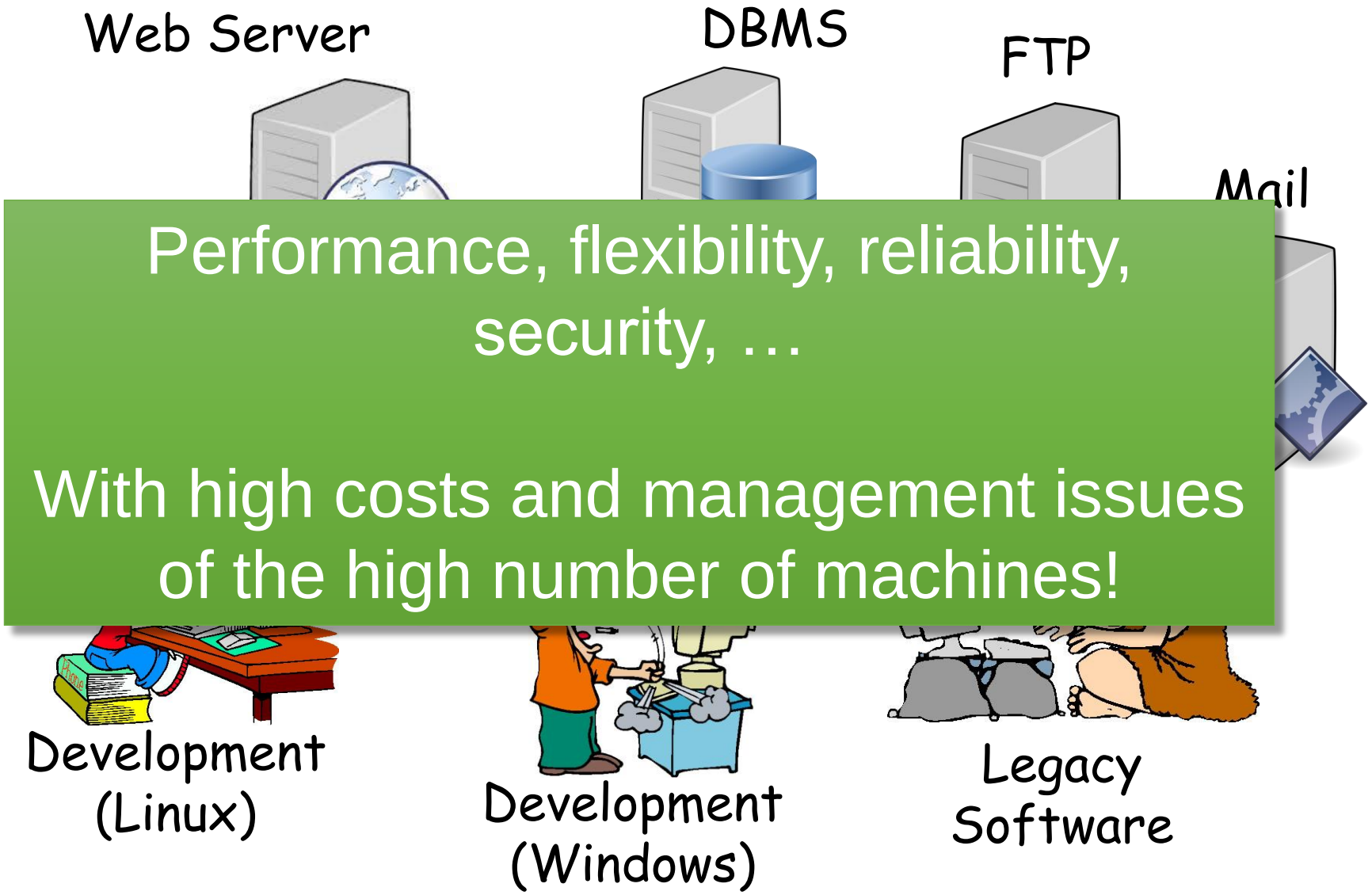
- References:
  - A. Tanenbaum, H. Bos, "*Modern Operating Systems*," Pearson ed., 4th ed, 2015 (Chapt. 7: Virtualization and the Cloud)
  - Neiger et al., "*Intel® Virtualization Technology: Hardware Support for Efficient Processor Virtualization*", Intel Technology Journal, 10(3), 2006
  - G. Buttazzo. "Hypervisors". Notes from the "Component-based software design" course
  - S. Xi, et al. "Real-Time Multi-Core Virtual Machine Scheduling in Xen"

Virtualization: overview

Virtual Machines

- A **Virtual Machine** (VM) is an emulation (using hw/sw techniques) of a real computing machine
- Every VM executes its own operating system and applications
- VMs are managed by a **virtual machine monitor** (VMM), or **hypervisor**
- The VMs running on a machine share the physical resources of the machine (processors, memory, I/O, …)

Physical machines: the past



Performance, flexibility, reliability, security, …

With high costs and management issues of the high number of machines!

Virtualized systems: the present

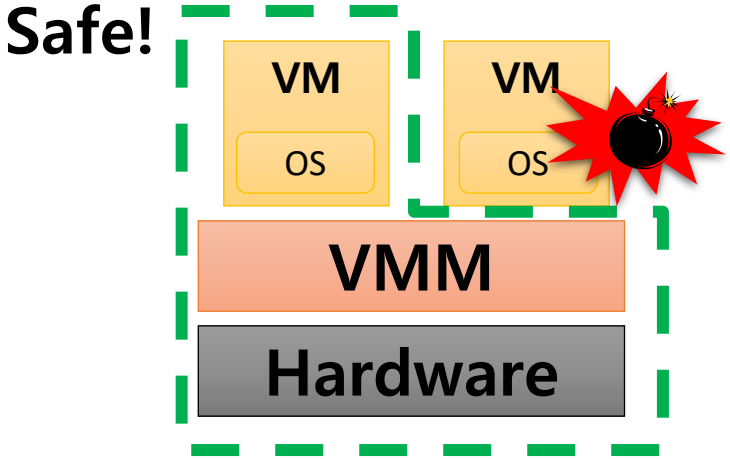
- Efficiency
- Consolidation
- Flexibility
- Cloud computing
  - *pay-per-use*

Virtualized real-time embedded systems… the future!

- Bringing virtualization concepts and advantages to embedded real-time systems!
- … not for free! Many technical challenges ahead.
- We'll be back on this topic later on ☺

Reliability and security

- Virtual machines guarantee a good degree of reliability and security
- The OS within the VM “sees” an **isolated** physical machine, and cannot damage the other VMs or the VMM
- The VMM is the unique privileged component (i.e., running in kernel mode) that can manage the physical hardware on behalf of VMs.



Interesting feature for Mixed-criticality systems!

Mixed-OS environments

- Multiple VMs can be implemented on a single hardware platform to provide individuals or user groups with their own OS environments

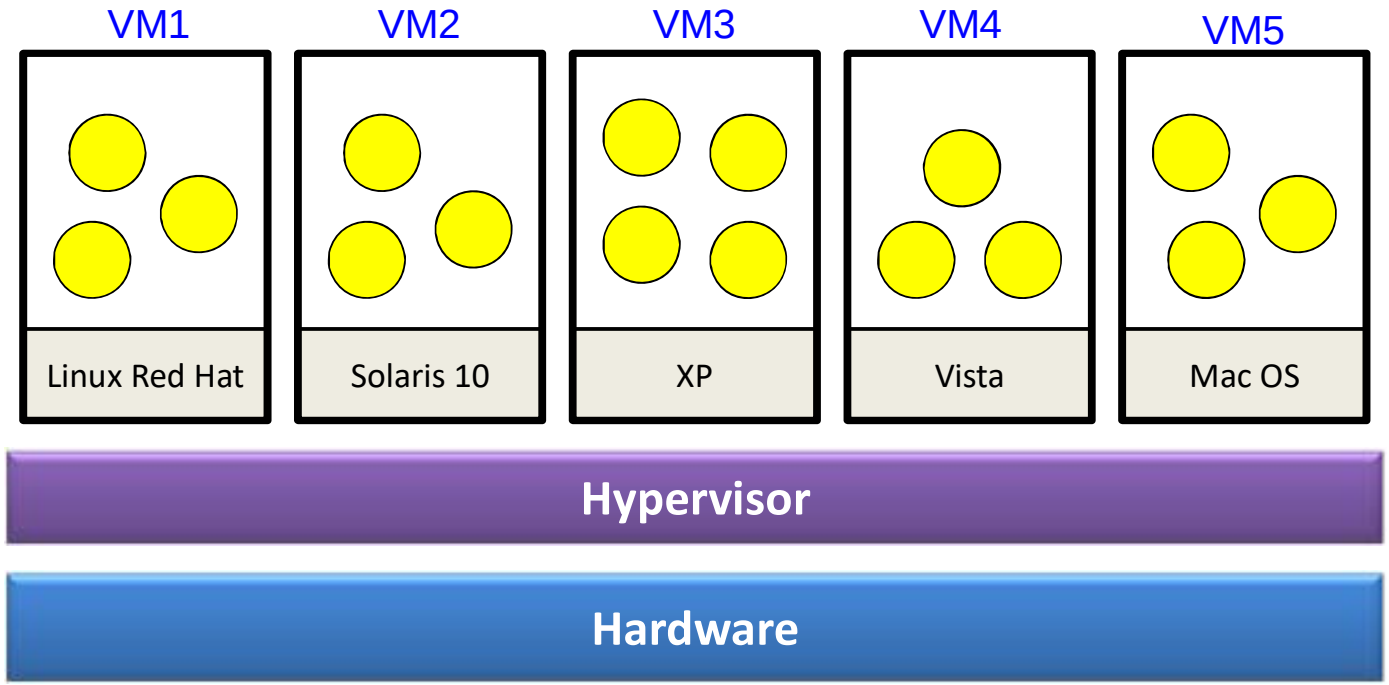
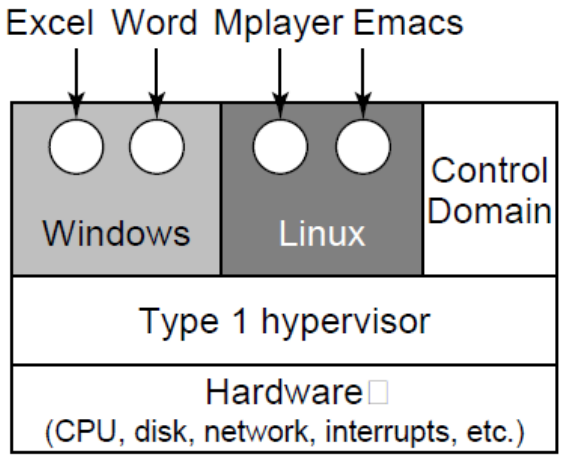
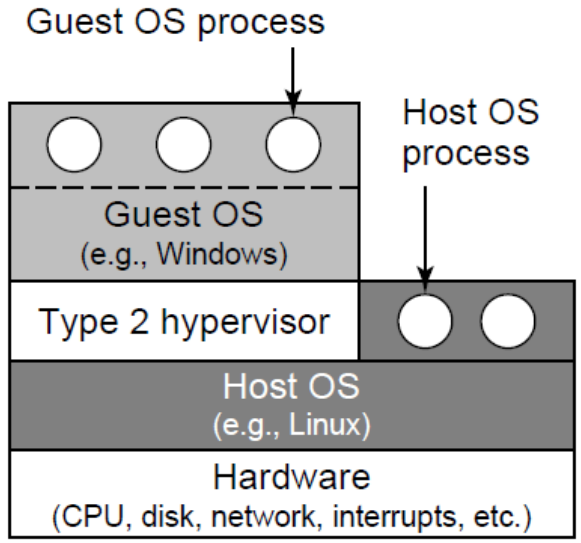


Figure: G. Kesden

Virtualization architectures





Type 1 Hypervisor

The VMM executes directly on the hardware

(**bare-metal virtualization**)

Type 2 Hypervisor

The VMM executes as an app on an OS (e.g. Windows)

(**hosted hypervisor**)

Virtualization architectures

- Type 1 architectures achieve better performance levels. It is the most suited solution in **server environments**
  - VMware ESXi; Xen; Hyper-V
- Type 2 architetures simplify the integration between the OS in the VM (**guest OS**) and the user's OS (**host OS**). It is the most suited solution in **desktop environments**
  - VMware Workstation/Fusion; Oracle VM VirtualBox

What do we have to virtualize?

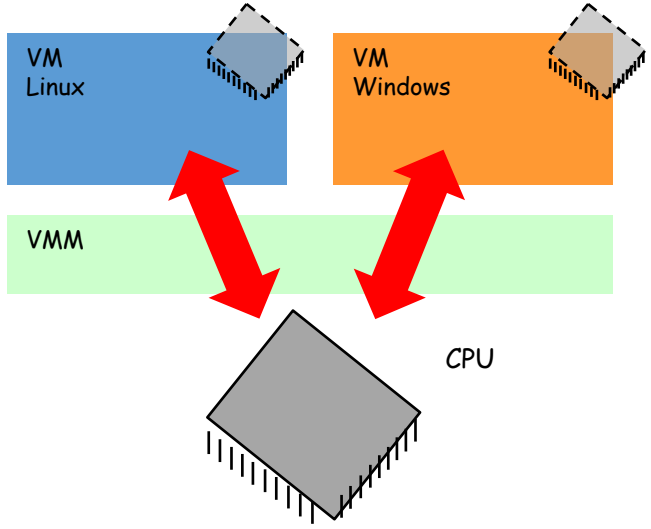
- **CPU**: provide vCPUs to virtual machines, with the abstraction of exclusive use of the CPUs; mapping vCPUs to pCPUs

- **Memory**: assign memory space to virtual machines, avoiding interference

- **Input/Output**: manage IO devices, considering competing requests coming from multiple virtual machines

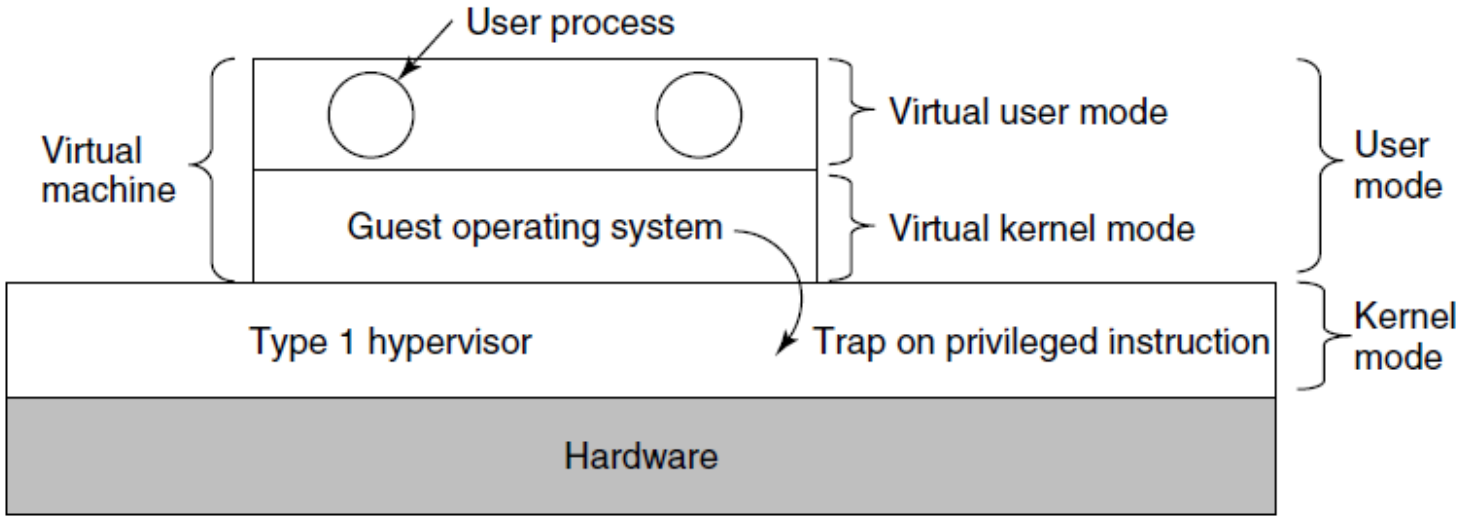
CPU virtualization

- The co-existence of multiple OS and of the VMM poses the problem of interferences when using the same CPU
- Traditional OS are developed assuming they have exlcusive and priviledged access to all resources
- The VMM must implement an emulation of resources, operating a so-called -de-privileging



De-privileging

- **De-privileging**: guest OSes and the VMM co-exist, executing at different levels of priviledge
  - The VMM plays the role previously occupied by the OS (**kernel/supervisor mode**)
  - The VM (guest OS + processes) execute in user mode, which in turn is divided in **virtual kernel model** and **virtual user mode**



Trap-and-emulate

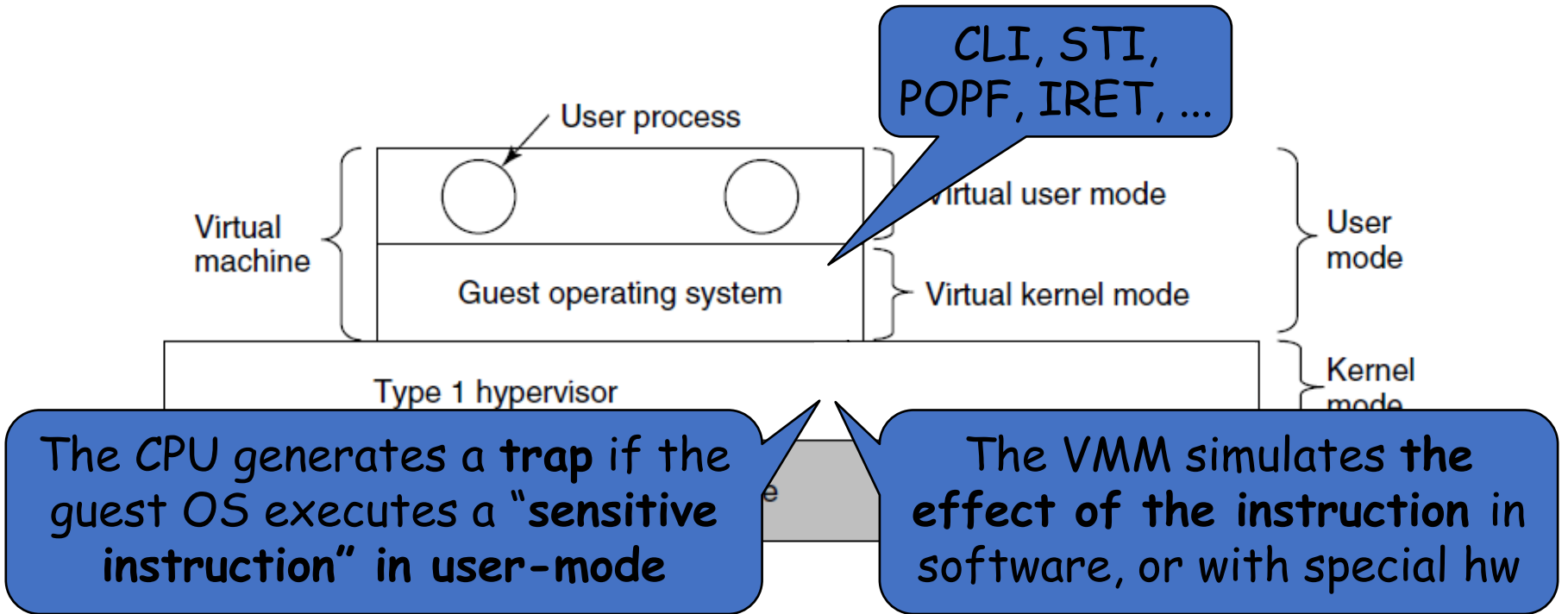
- Popek and Goldberg, 1974

- “For any conventional third-generation computer, an *effective* VMM may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.”

- In other words… It is sufficient that all the instructions that could affect the correct functioning of the VMM (**sensitive instructions**) always **trap** and pass control to the VMM.

Trap-and-emulate

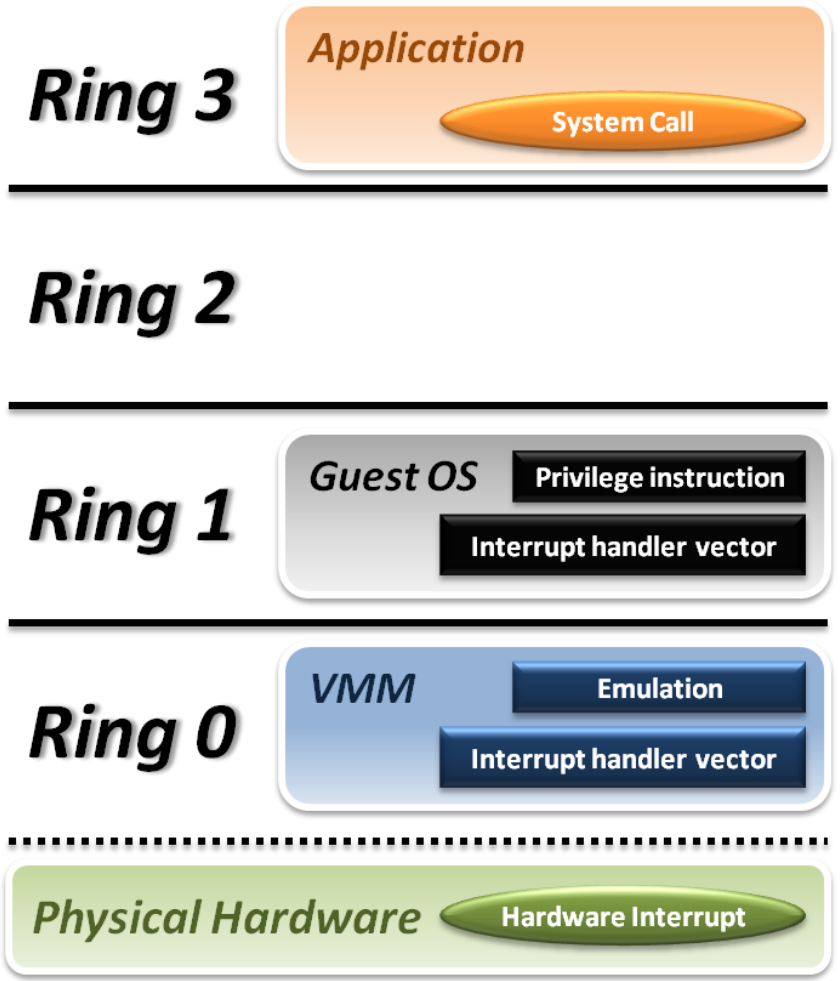
- If the guest OS tries to use a sensitive instruction, such as:
  - Management of I/O registres, MMU, interrupt, CPU status, …
- …The VMM intercepts and emulate such instructions (**trap-and-emulate**) keeping control of the physical CPU



Ring de-privileging: the example of Intel CPUs

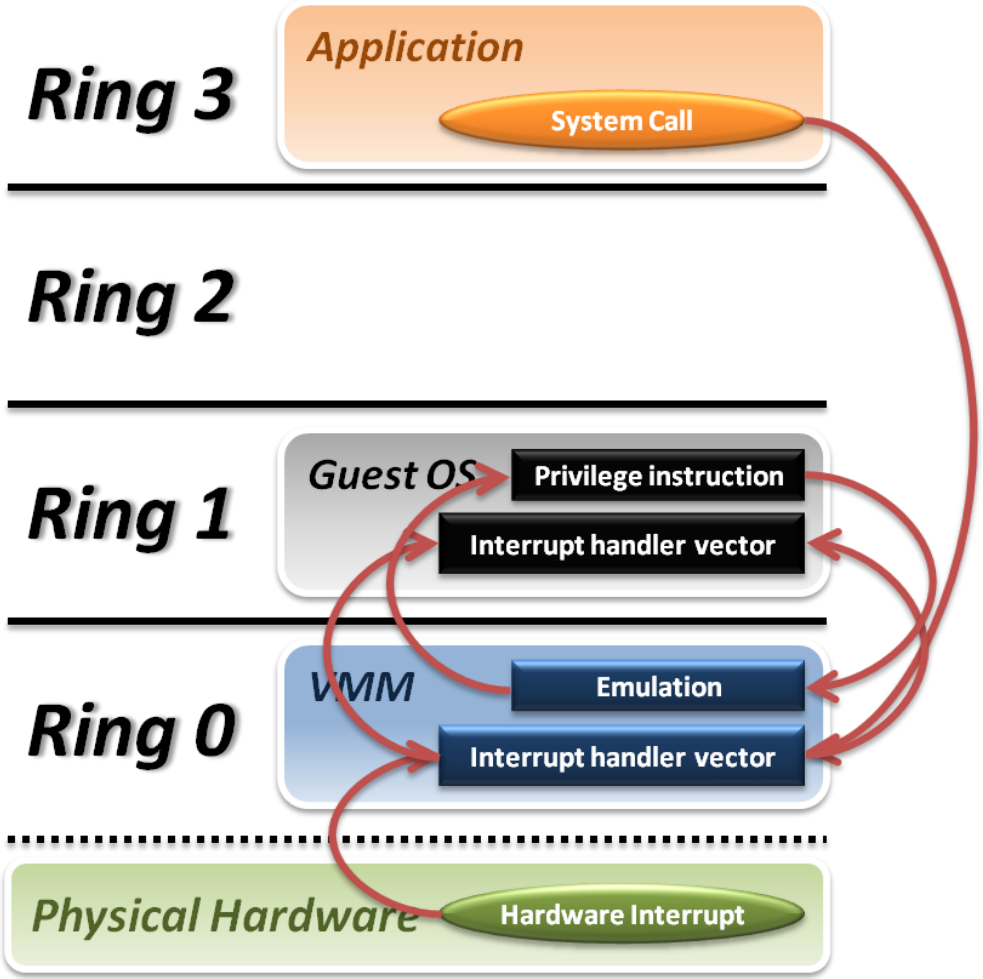
- Four levels of protection present to isolare and protect user programs from other OS programs

- Every task executed by the CPU is executed according to the *Current Privilege Level*, which value can be variated also trough sensible instructions



Ring de-privileging: the example of Intel CPUs

- **System call**
  - The CPU generates a trap managed by the VMM
  - The VMM jumps to the guest OS
- **Hardware interrupt**
  - The hardware interrupt is served by the ISR within the VMM
  - The VMM jumps to the correspondent ISR of the guest OS
- **Privileged instructions**
  - The execution of a privileged instruction in the guest OS generates a trap managed by the VMM
  - The VMM emulates the instruction and jumps back to the guest OS



The (non) “virtualizability” of Intel CPUs

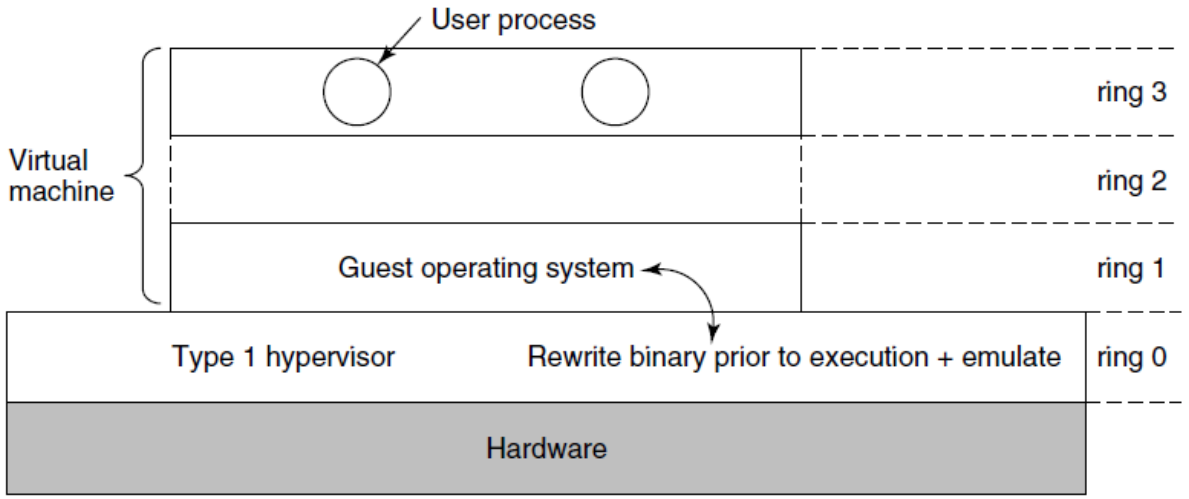
- The traditional x86 Intel architecture **is not** “**virtualizable**” using trap-and-emulate
- In x86, many sensible instructions simply **do not generate any trap**!
  - If a user process or the guest OS try to execute the sensitive instruction, the CPU just **ignores the instruction** (without generating any trap)
  - These instructions are called “sensible but not privileged”

CPU virtualization techniques

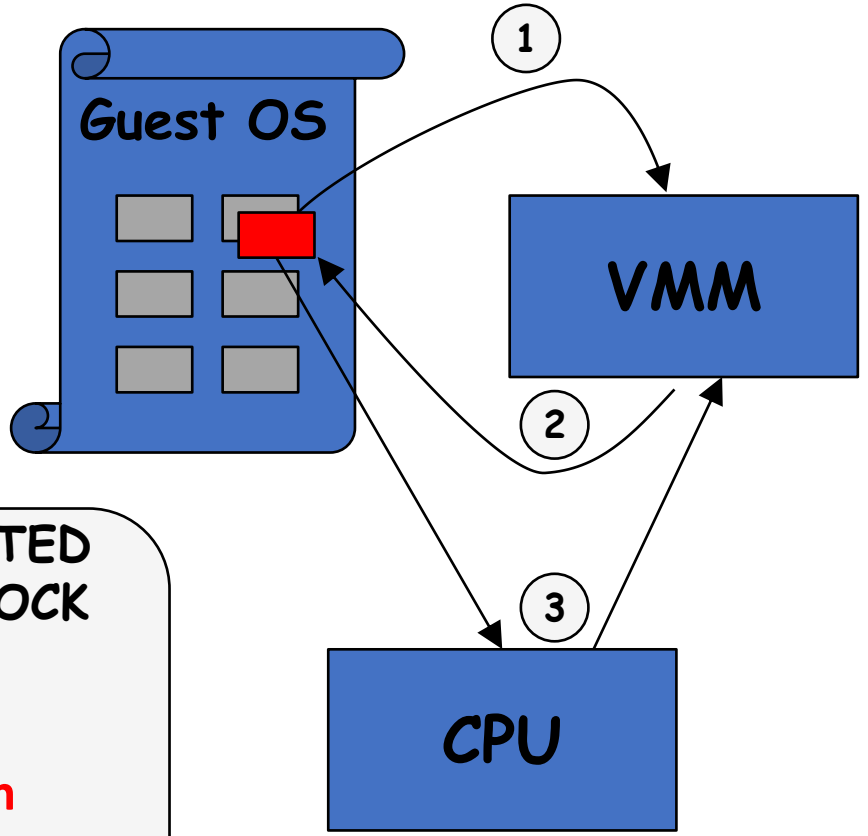
- **Full virtualization**, without hardware support
  - Needs software techniques
    - (dynamic binary translation, shadow page tables, ...)
  - Example hypervisors: VMWare, QEMU
- **Para-virtualization** Hypercall
  - The guest OS is “re-written” to cooperate with the VMM
  - Example hypervisors: XEN
- **Full virtualization**, with **hardware support** (Intel VT, AMD SVM, Armv8-A AArch64)
  - Improved performance and simplified VMM
  - Example hypervisors: XEN, VMWare, KVM

Full virtualization, without hardware support

- In 1999, the VMware hypervisor introduced efficient full-virtualization techniques for Intel x86
- The **binary code** of the guest OS is **rewritten** “on-the-flight” by the VMM, replacing sensible instructions with an emulation code
  - Dynamic binary translation



Dynamic binary translation



BASIC BLOCK

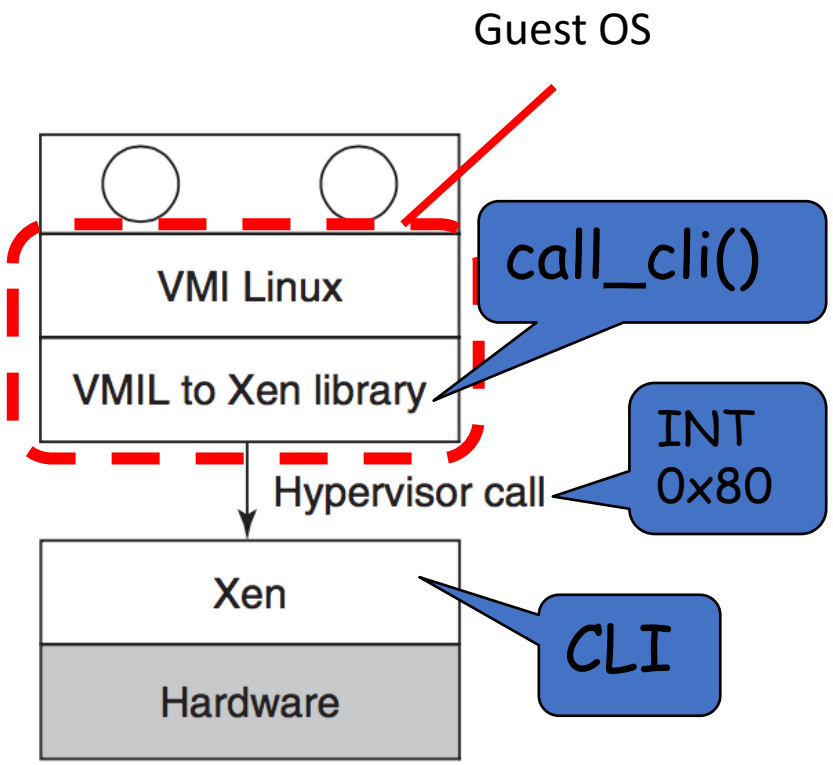
mov ...
add ...
cli
add ...
...
jmp <guest>

TRANSLATED BASIC BLOCK

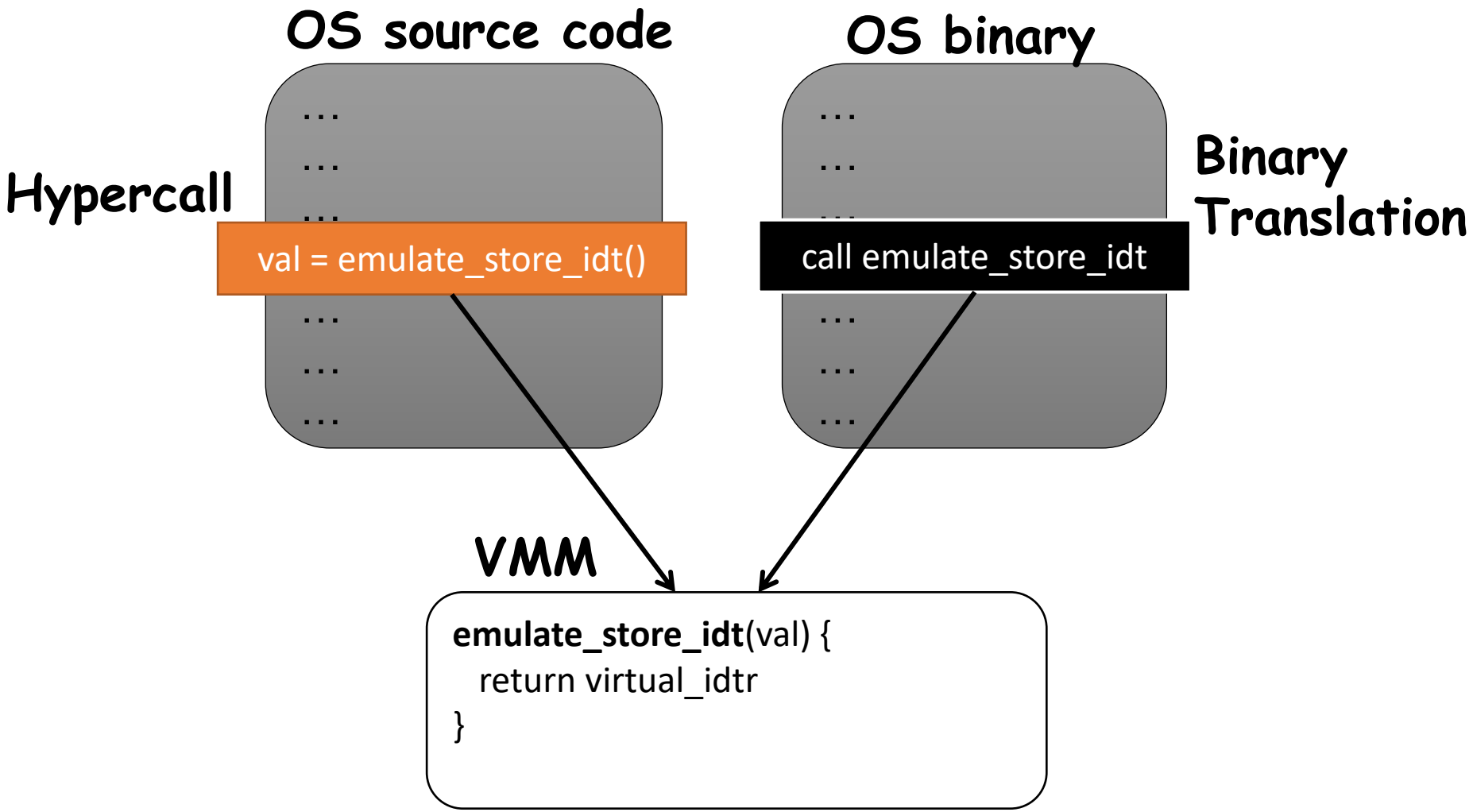
mov ...
add ...
cli emulation
add ...
...
jmp <VMM>

Para-virtualization

- The VMM exposes a software interface to the guest OS in the form of **hypercalls**, called by the guest OS instead of sensible instructions
  - Xen is one of the most representative hypervisor of this category
- It requires that the guest OS is rewritten in order to execute on the VMM
  - The guest OS is aware of the VMM
- Not applicable to legacy or proprietary systems, such as Windows



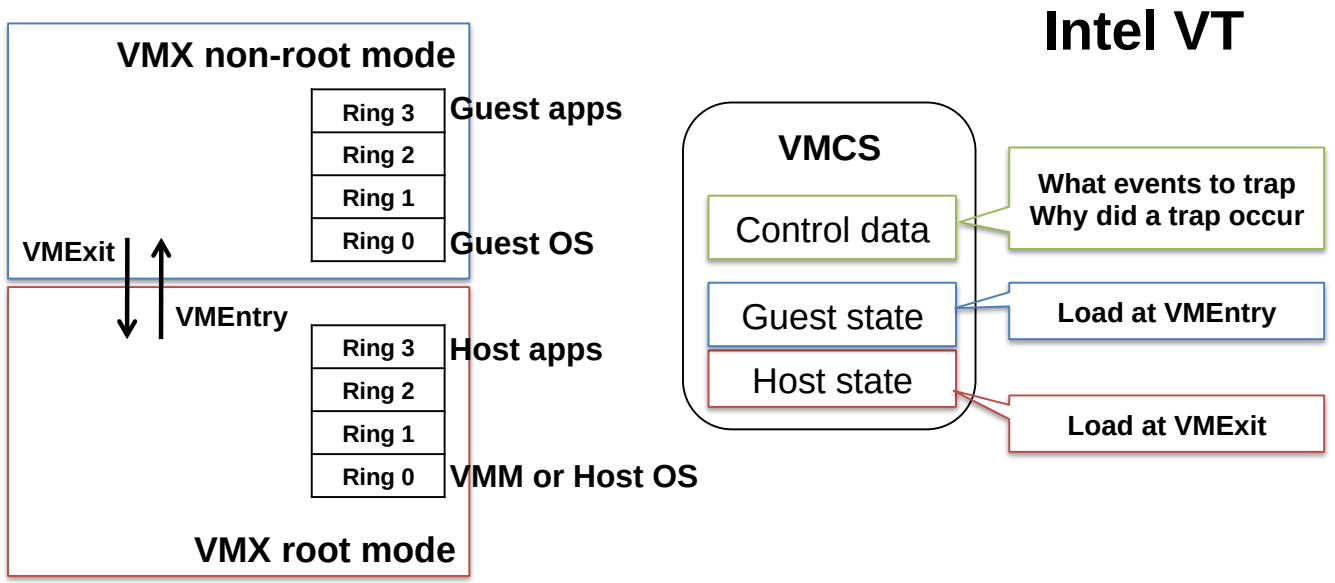
Hypercall vs. DBT



Hardware support for CPU virtualization

- Intel VT introduces **VMX root/non-root modes** and **VMCS** (VM Control Structure)
  - **Simplify** the VMM, a good portion of virtualization operations are performed in hardware
  - **Avoid** guest OS **ring de-privileging**
  - Allow a flexible configuration of the trap mechanism at single instruction level
  - Improve the efficiency of the context switch between VMs and the VMM

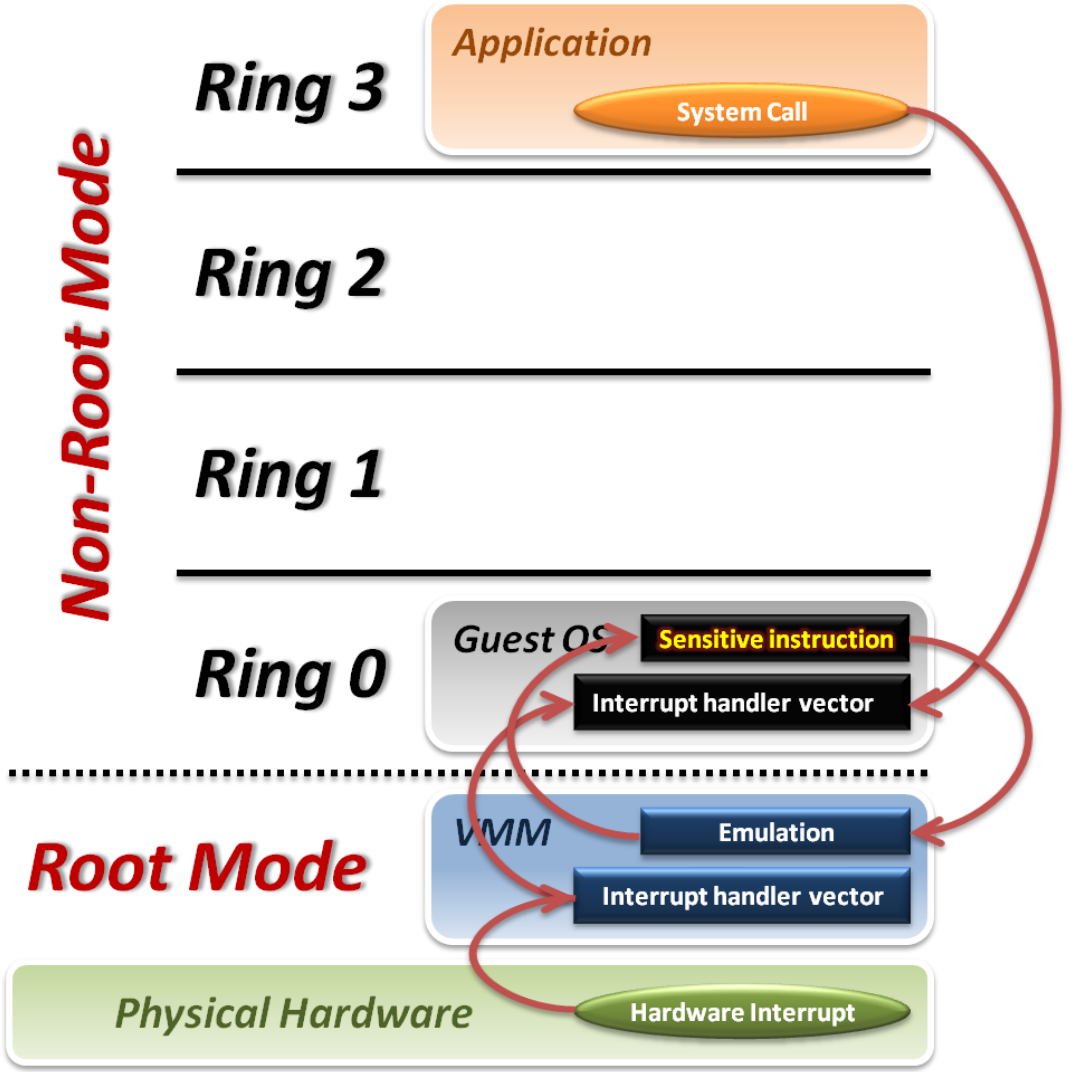
Hardware support for CPU virtualization



- **VMX root** mode (execution of the **VMM**)
  - All the instructions are executed in the same mode of a traditional CPU
- **VMX non-root** mode (execution of the **guest OS**)
  - The behavior of sensible instructions is re-defined by the VMM
- **VM-exit**, **VM-enter**: events that cause the passage between root/non-root modes (specified in the **VMCS**)

Hardware support for CPU virtualization

- **System call**
  - The trap is directly managed by the guest OS ISR
- **Hardware interrupts**
  - Hardware events are managed by the VMM and redirected to the Guest OS, when necessary
- **Privileged instructions**
  - Sensitive instructions generate traps that are emulated by the VMM

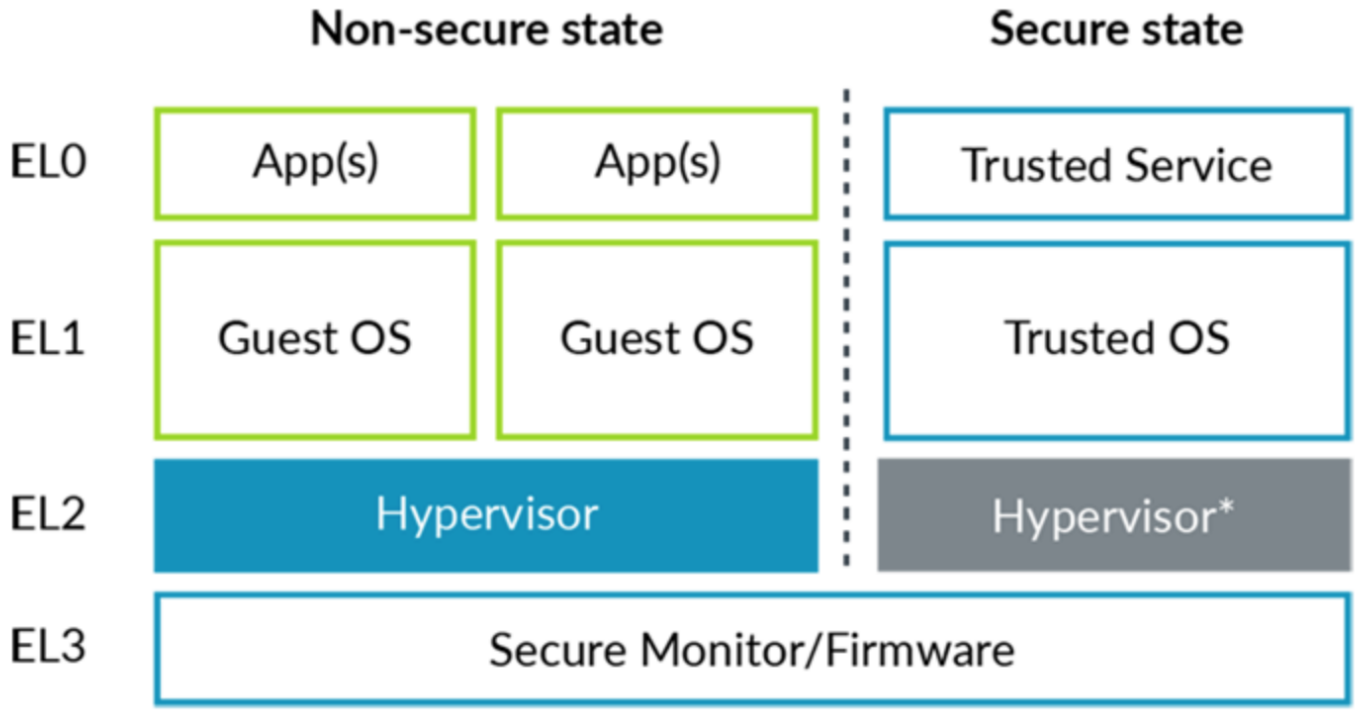


VM-exit causes examples

- Sensible instructions
  - **CPUID**: reports CPU capabilities
  - **RDMSR**, **WRMSR**: read/write “model-specific registers”
  - **INVLPG**: invalidates the TLB
  - **RDPMC**, **RDTSC**: reads performance monitoring and timestamp registers
  - **HLT**, **MWAIT**, **PAUSE**: disactivation of the guest OS
  - **VMCALL**: new instruction introduced to invoke the VMM
- Accesss to sensible state
  - **MOV DRx**: access to debug registers
  - **MOV CRx**: access to control registers
  - **Task switch**: access to CR3 (pointer to the page table)
- Exceptions and asynchronous events
  - Page fault, debug exceptions, interrupts, etc.

ARMv8-A virtualization support

- EL is the Exception Level: EL0/1 register access generate traps
- EL2 not always supported in the secure state (Trust Zone)

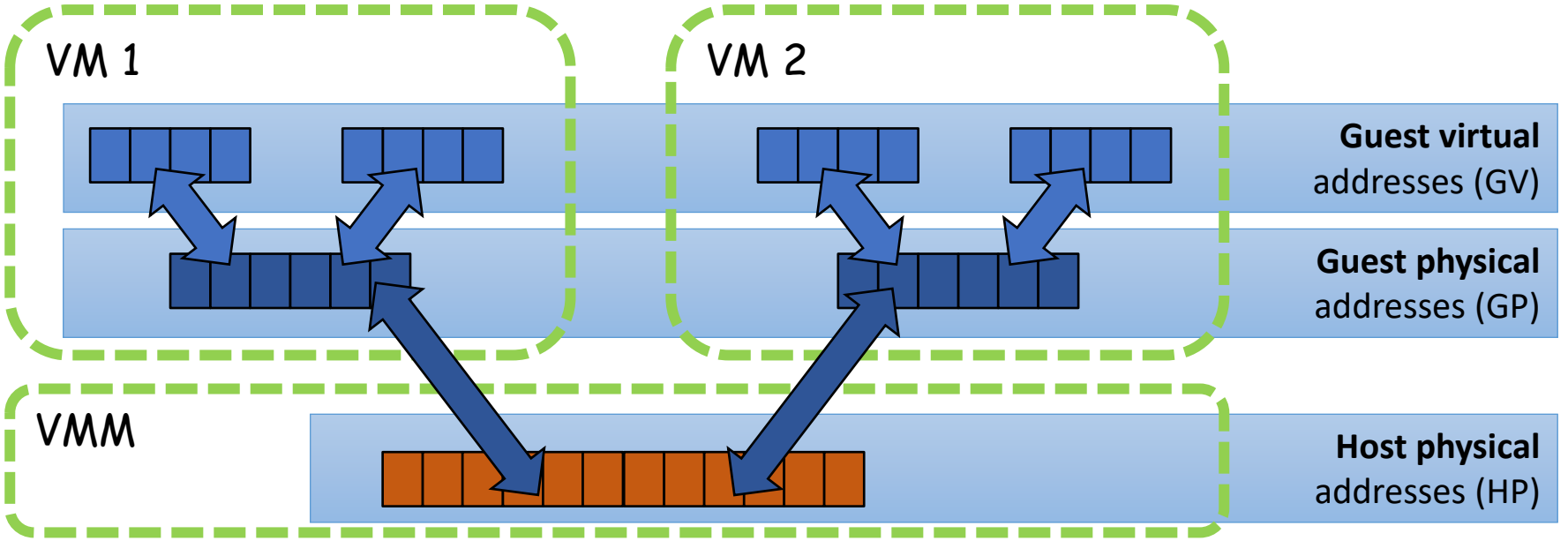


More info at: https://developer.arm.com/documentation/102142/latest/

Memory virtualization

- Inside a VM, the guest OS manages the “fake” physical memory of the VM, using traditional virtual memory mechanisms
- The VMM manages the “actual” physical memory, trough demand paging, swapping, etc.
- VMs are isolated and cannot access the memory of other VMs

Virtual – Physical addresses



How to realize the mapping?

How to realize the mapping

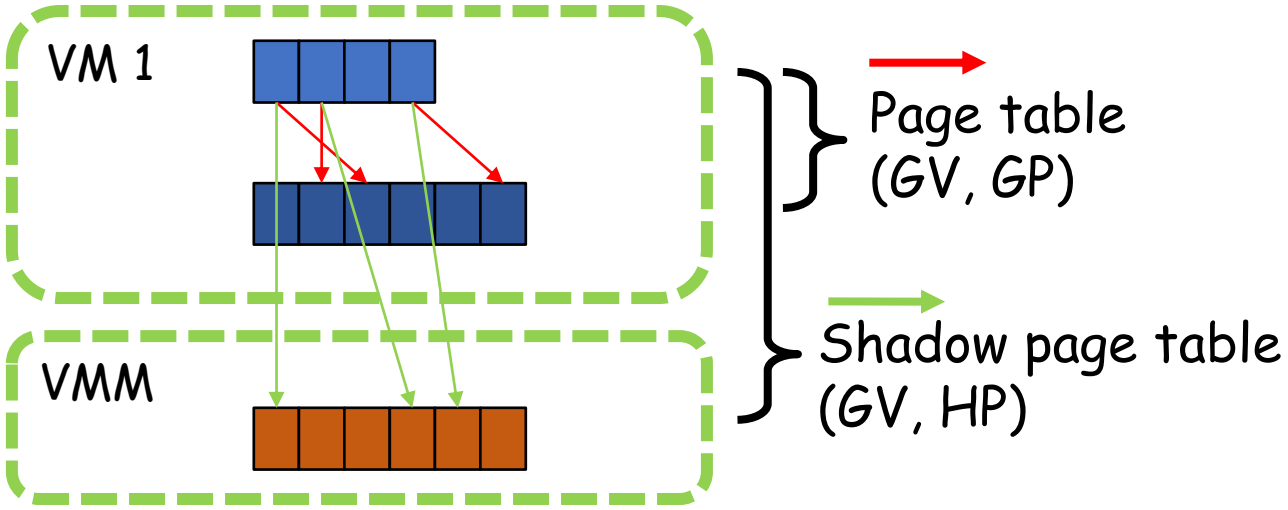
- Shadow page tables: No hardware support
  - Software-based approach realized by the VMM

- Extended page tables (EPT): virtualization support
  - Hw feature introduced by Intel VT for HW-Assisted virtualization

Shadow page tables

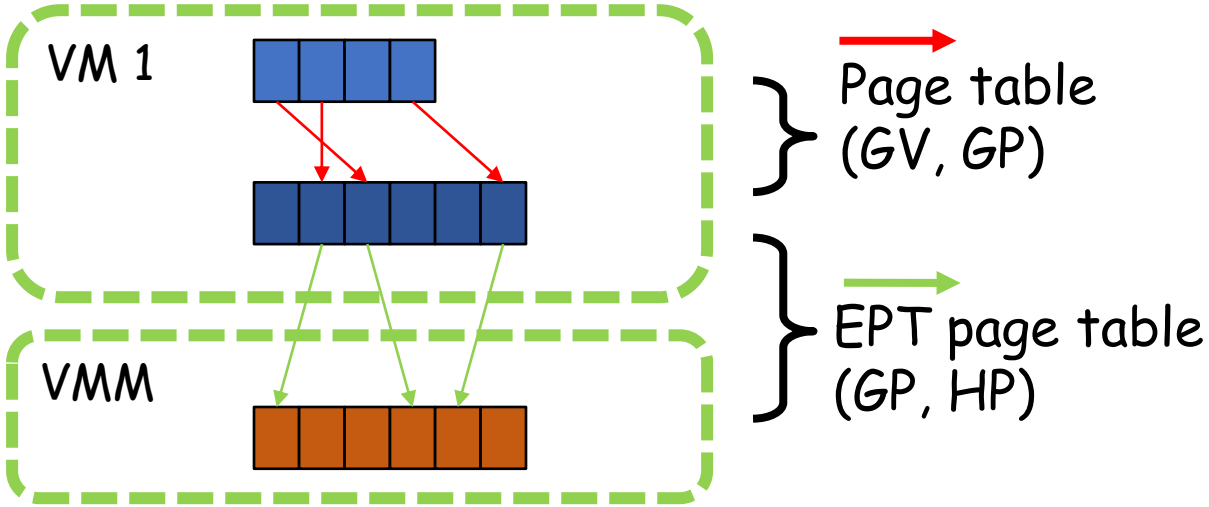
- For each **page table** of the guest OS, the VMM creates a **shadow page table**
- The VMM virtualizes the *page-table base register* (**PTBR**), and intercepts the guest OS when it tries to modify a page table
- the VMM decides the allocation of physical pages, and updates the shadow page table
- The CPU uses the shadow page table, and ignores the page tables of the guest OS
  - **KEY POINT: does not require hardware features, but introduces software overhead!**

Da GV a GP abbiamo page fault che poi viene risolto dal VMM



Extended page tables

- Intel VT hardware support to memory virtualization
- the CPU manages directly the page tables of VMs
- hypervisor-induced page faults are avoided
- Requires however a double translation GV→GP and GP→HP
  - Average performance improvement of 48% for MMU-intensive benchmarks, even if can increase memory access latencies for some workloads



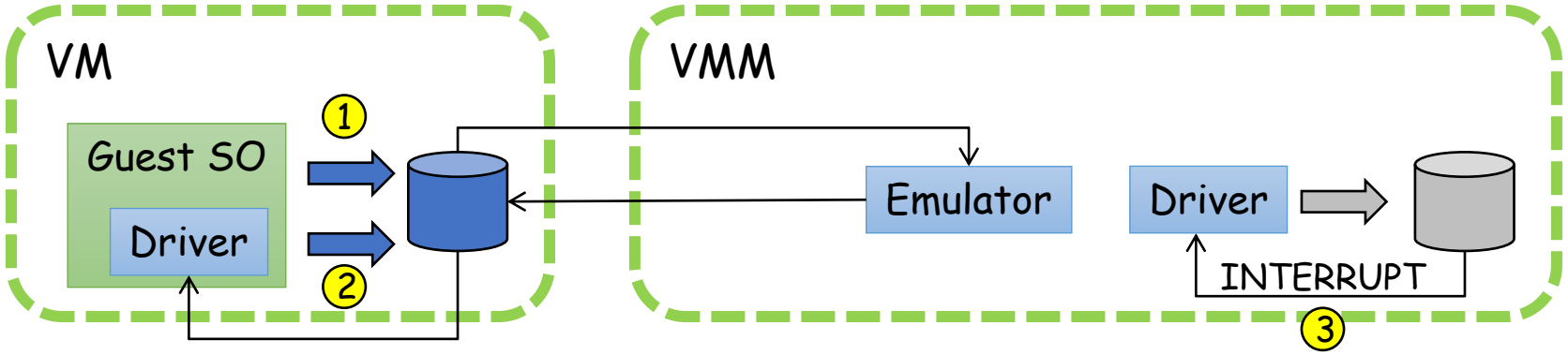
I/O virtualization

- I/O instructions (programmed I/O, memory-mapped I/O, DMA) are “virtualizable”
- The VMM can intercept them *trap-and-emulate*, and simulate a virtual device, e.g.:
  - **Virtual disk**: the VMM decodes the read/write command and executes it on a file dedicated to the virtual disk
  - **Virtual networking**: the VMM decodes the frame and forwards it to the physical or virtual net (**virtual switch**)

I/O full virtualization (trap and emulate)

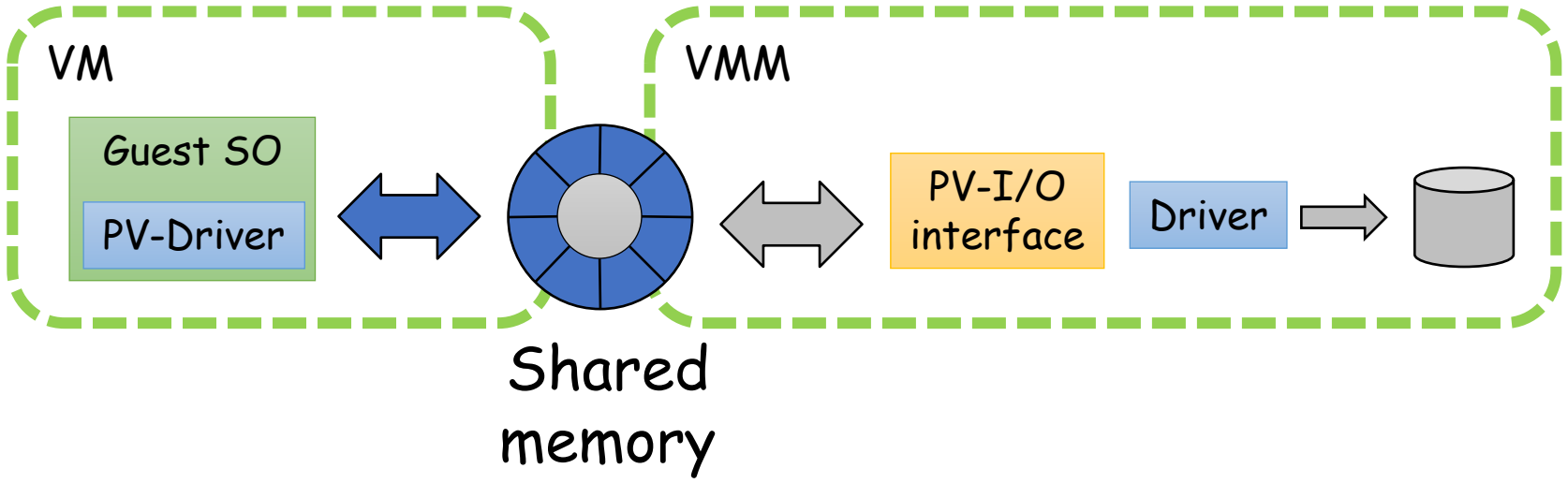
1. The guest OS tries to interrogate the bus to “open” a device; the VMM intercepts the instruction and answers with the reference to a virtual device (e.g., emulated with QEMU)
2. The guest OS tries to read/write the registers of the virtual device; the VMM intercepts and copies the values from/to the real hardware registers
3. The real hardware device sends an interrupt; the VMM serves it and then simulates (injects) an interrupt in the VM context, that will be served by the local driver

- Nice technique to hide and emulate devices, but introduces VMM overhead!



I/O paravirtualization

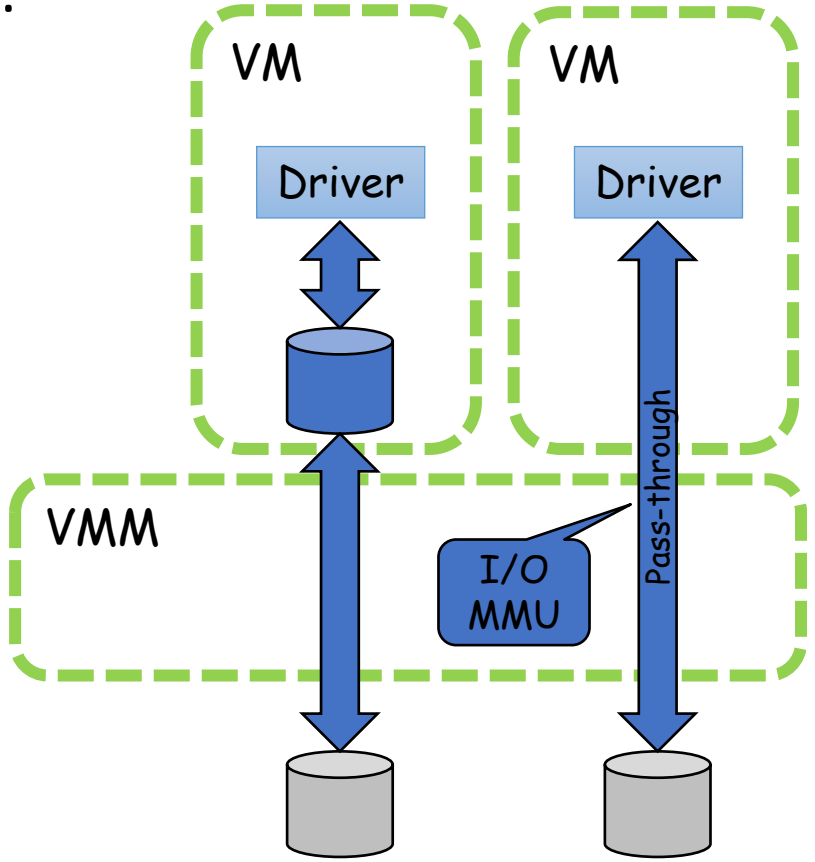
- The VMM shares a memory area with the guest OS and provides special I/O instruction
- The guest OS is aware of the virtualization and uses a proper “paravirtualized” driver, that reads/writes in the shared memory
- The VM translates operations on the shared memory with operations on the real device, with a PV-I/O driver
- Improved performance since no device emulation is required



Device pass-through, I/O-MMU, SR-IOV

- Device pass-through: an I/O device is **exclusively** assigned to a VM, improving the overall performance (but inhibiting efficient device sharing among VMs)
- I/O-MMU: **memory-mapped I/O** operation:
  - Guest OS communicate its GP addresses to the device's DMA
  - the DMA tries to write in GP addresses that are however translated to HP addresses by the I/O-MMU
  - I/O-MMU also performs interrupt remapping to forward device interrupts to the VM
- An alternative: SR-IOV
  - Single Root I/O virtualization
  - An hardware feature (supported by PCIe) that allows to share a device without a VMM
  - Exposes virtual device interfaces to VMs

Il VMM come se non ci fosse



Why virtualization in real-time systems?

Why virtualization in real-time systems?

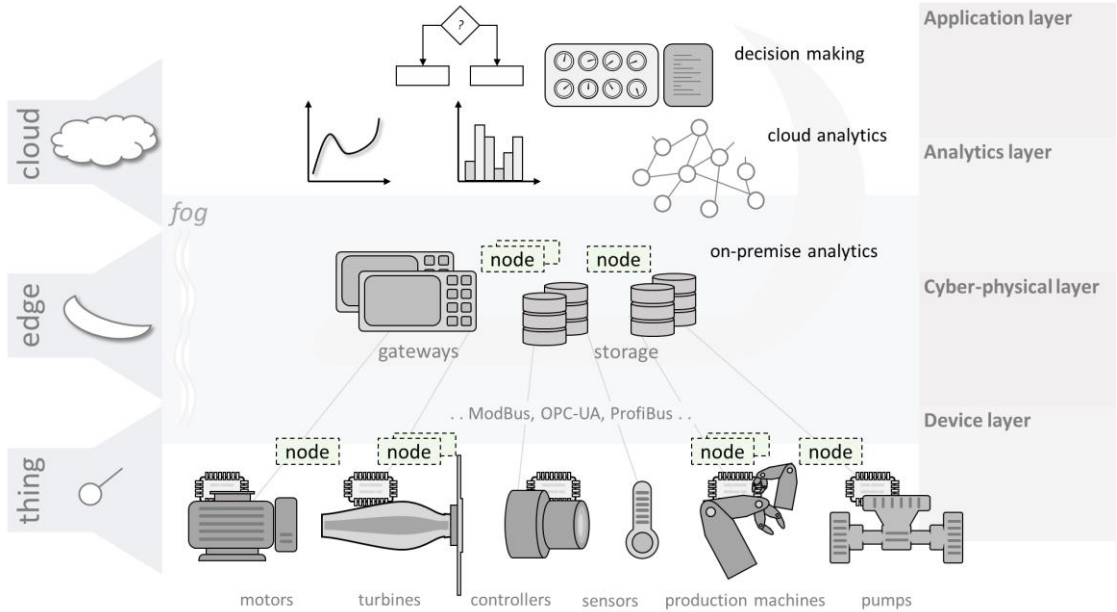
- Virtualization is gaining traction in real-time, embedded domains, due to the increasing interest in mixed-criticalty systems
- VMs can be seen as isolated environments, each with different requirements and criticality levels
- Good approach for incremental development and certification

Why virtualization in real-time systems?

- License separation
  - Systems composed by Linux + a proprietary software
  - The hypervisor can isolate Linux from the proprietary (critical) side
- Software architecture abstraction
  - Support for product series: same software running on different hardware
  - Decoupling from the real hardware
  - Benefits: time-to-market and engineering cost
- Certfication issues
  - Encapsulation of a safety-critical subsystem that can be certified independently of the other subsystems running on the same platform

Why virtualization in real-time systems?

- The Industrial IoT case
  - A multi-layered MCS
  - Differentiated real-time and reliability requirements
    - Latency
    - Delivery guarantees
    - Event time consistency

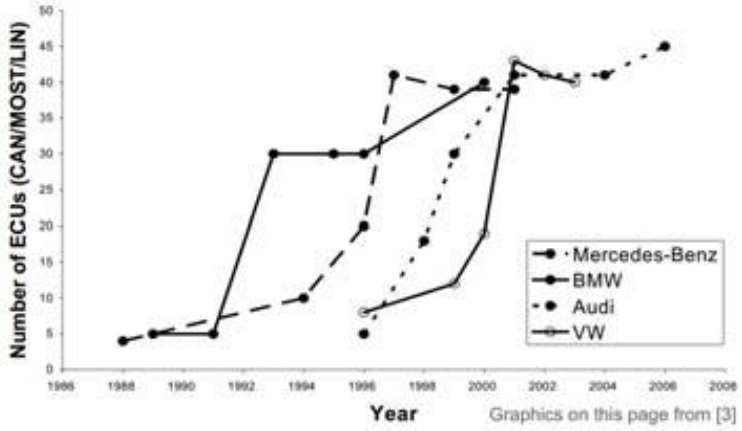


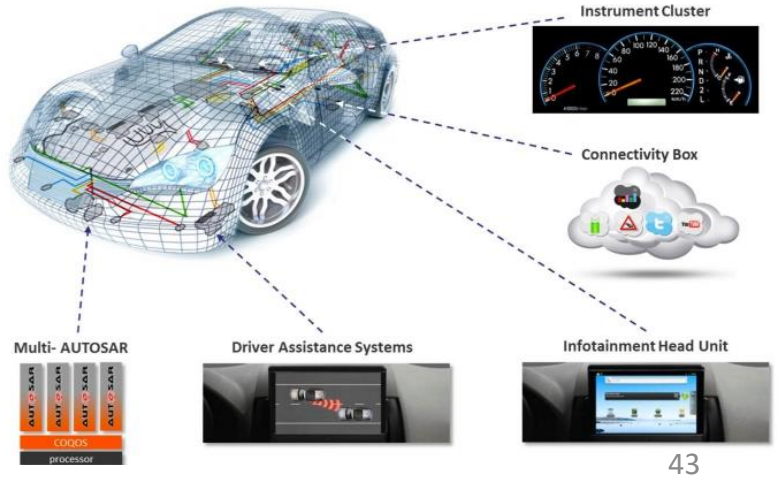
	at-least-once	best-effort
milliseconds	e.g., Emergency response	e.g., Real-time monitoring
hours	e.g., Predictive maintenance	

Source: C. Lu, “Real-time cloud computing”, http://www.cse.wustl.edu/~lu/

Why virtualization in real-time systems?

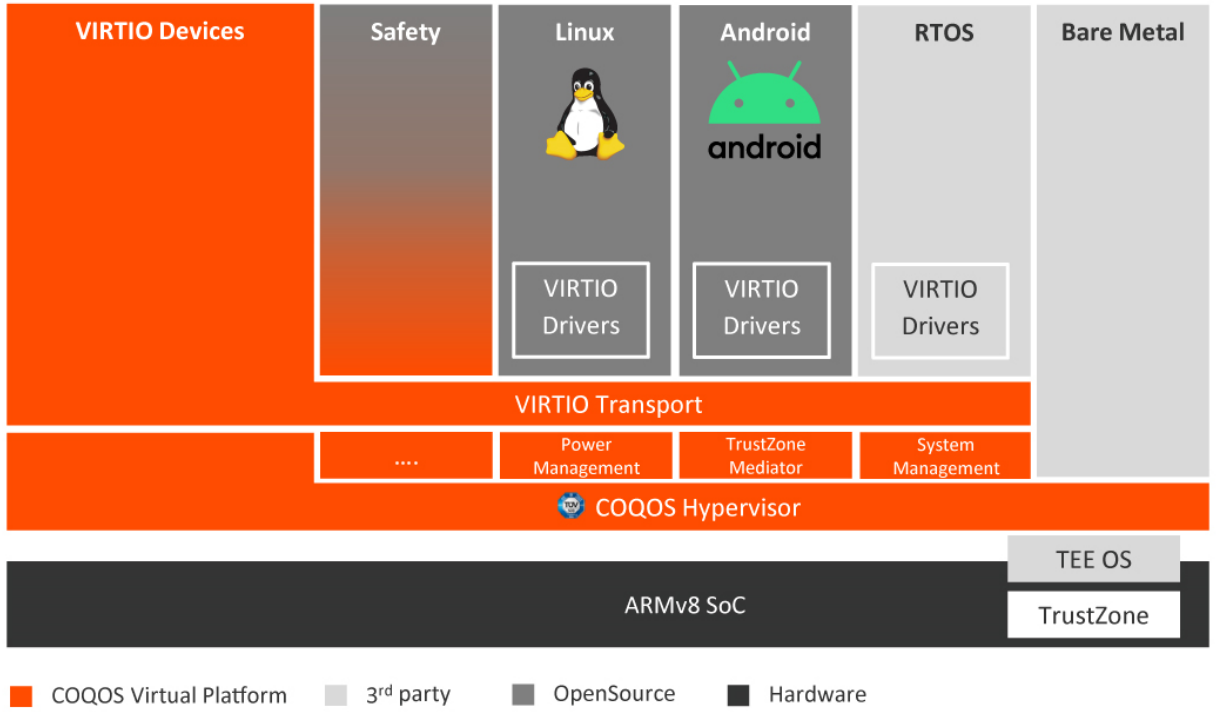
- The automotive case
  - Proliferation of ECUs, more than doubled in 10 years
  - The current challenge is to consolidate 100 ECUs on about 10 multicore processors
  - Integrating multiple systems on a common platform
    - Infotainment on Android
    - Safety-critical control on AUTOSAR OS





Source: http://www.edn.com/design/automotive/4399434/Multicore-and-virtualization-in-automotive-environments

Virtualization in embedded real-time systems



VIRTUALIZATION ON MICROCONTROLLERS

October 16, 2018 // By Stefaan Sonck Thiebaut, OpenSynergy

Email print Share Share reddit

Embedded virtualization is a key technology for the future of automotive. Virtualization makes it possible to allocate the resources of a processor to multiple safely separated applications and operating systems. This is an effective approach to redesign the vehicle electronics architecture, take full advantage of the performance of processors and address the growing complexity of software-defined functions.

Real-time hypervisors

Real-time hypervisors: challenges

- How to deal with latencies?
  - Similar issue of real-time kernels: non-preemptable sections in the VMM
  - Dealt with minimal design:
    - Separation kernels and Microkernels, such as SEL4-derived hypervisors
  - Or with preemptable hypervisors
    - E.g., KVM or XEN with PREEMPT_RT?
    - From an empirical study[1]:

Kernels	Core Duo		Core i7	
	Xen	KVM	Xen	KVM
NRT/NRT	$3216\mu s$	$851\mu s$	$785\mu s$	$275\mu s$
NRT/RT	$4152\mu s$	$463\mu s$	$1589\mu s$	$243\mu s$
RT/NRT	$3232\mu s$	$233\mu s$	$791\mu s$	$99\mu s$
RT/RT	$3956\mu s$	$71\mu s$	$1541\mu s$	$72\mu s$

[1] L. Abeni, D. Faggioli. “An experimental analysis of XEN and KVM latencies”. At ISORC 2019

Real-time hypervisors: challenges

- Global scheduler: how to schedule vCPUs assigned to VMs?

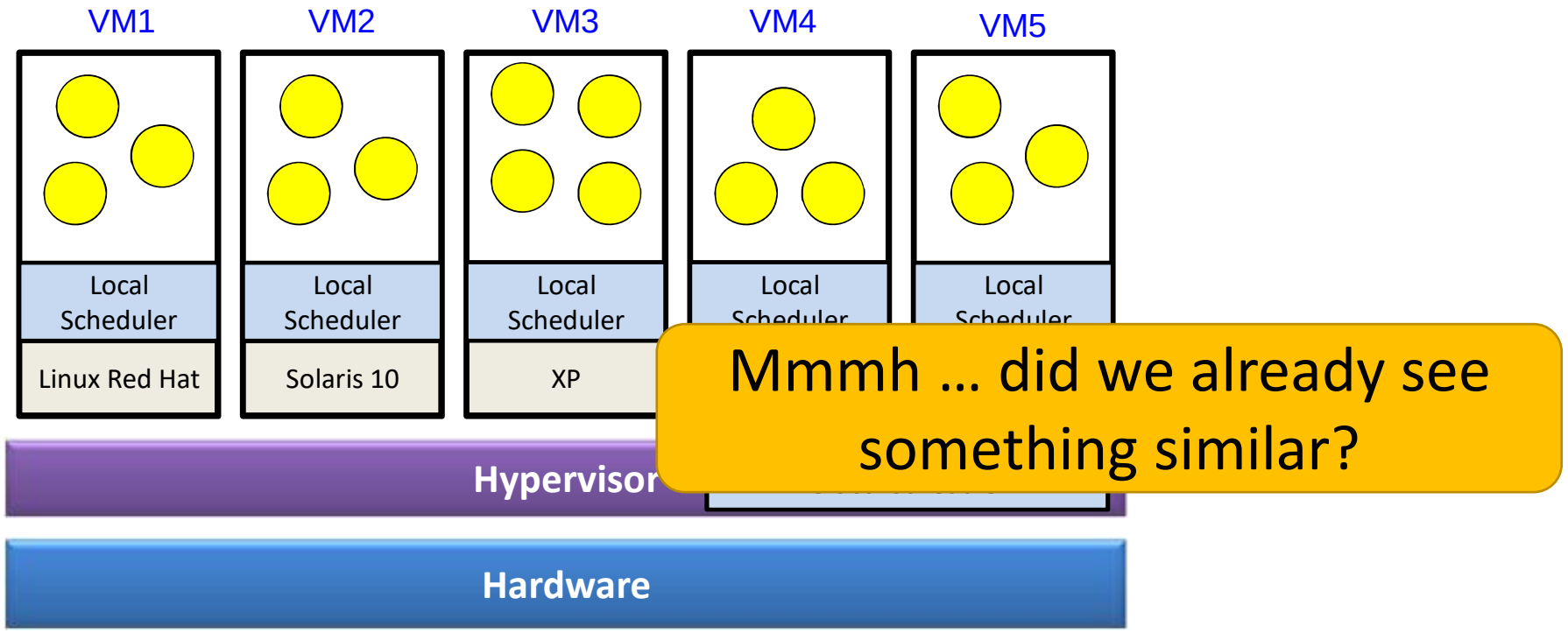
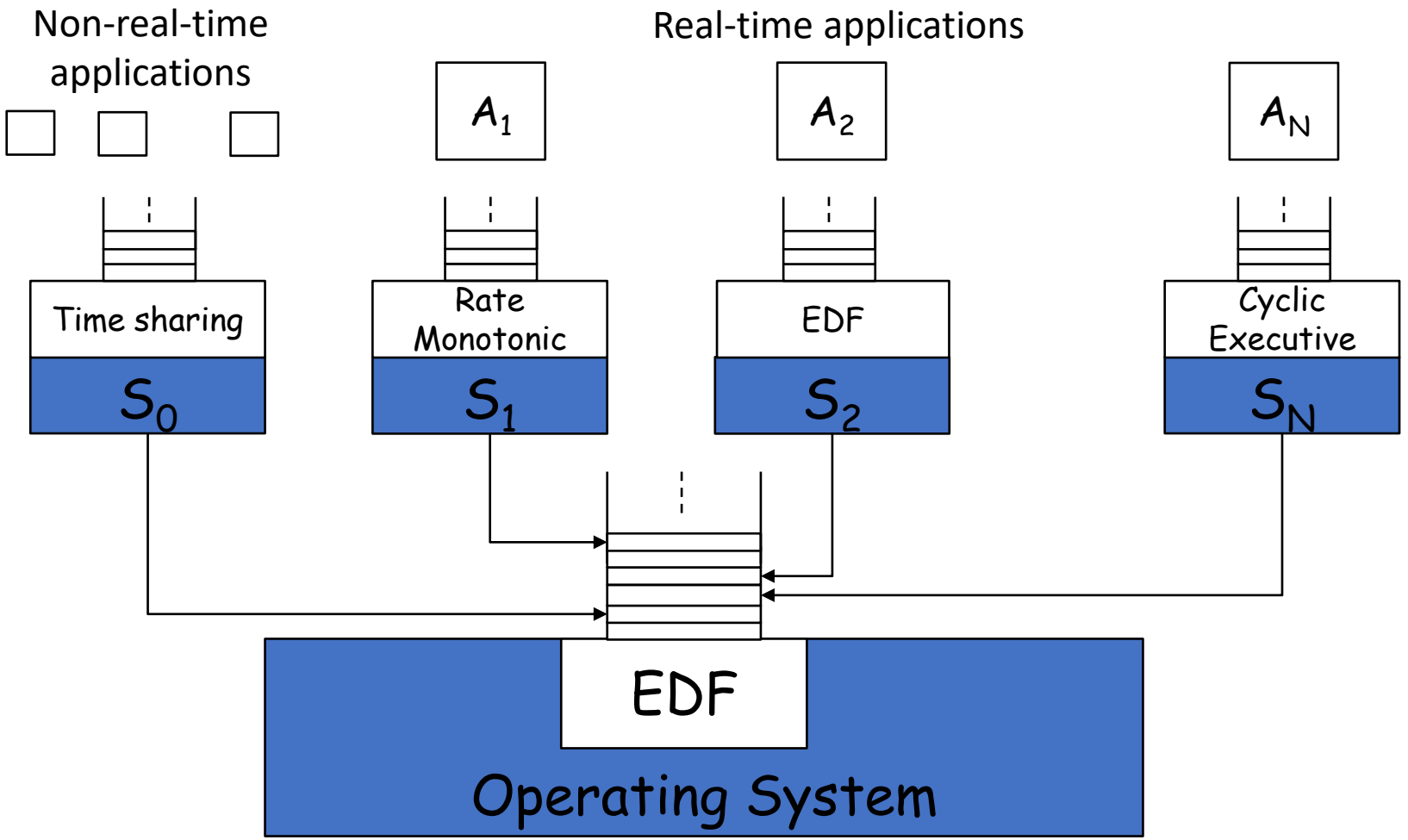


Figure: G. Kesden

Open system architecture



Open system architecture

- The Mixed-OS environment is an instance, based on virtualization and hypervisors, of the open system architecture!
- virtualization implies a **two-level hierarchical scheduling** system

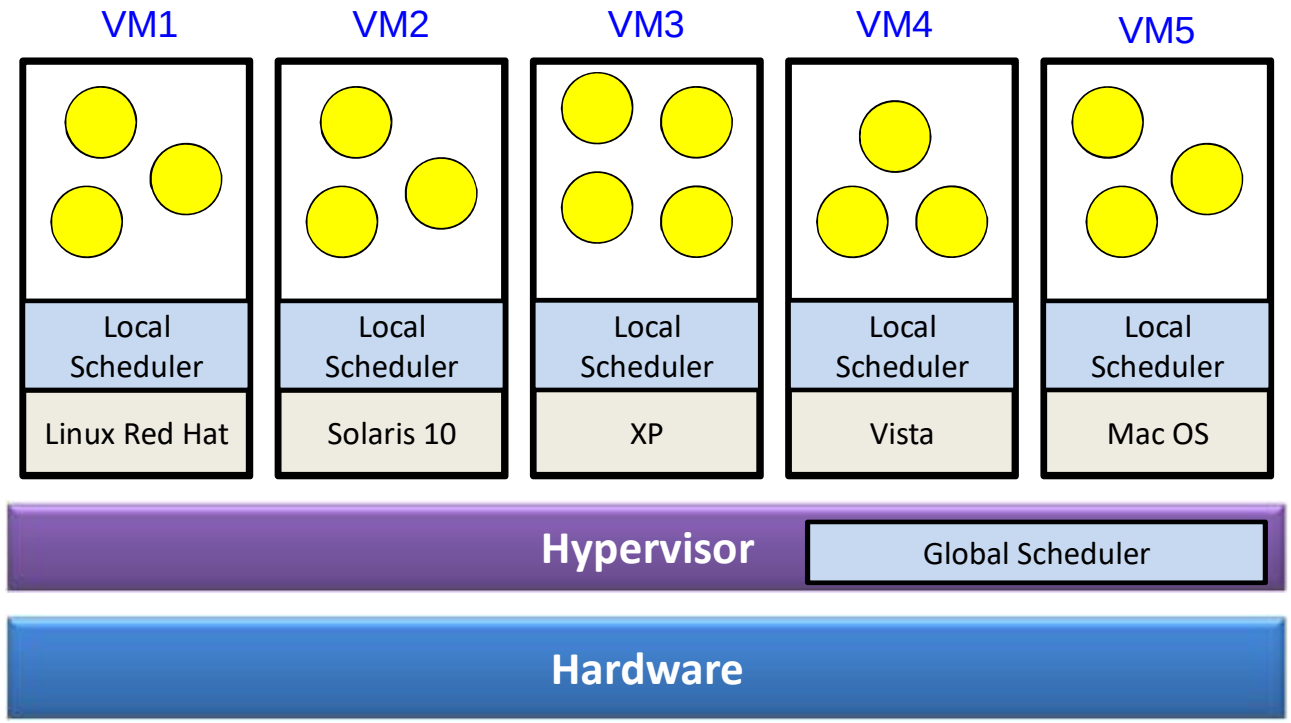


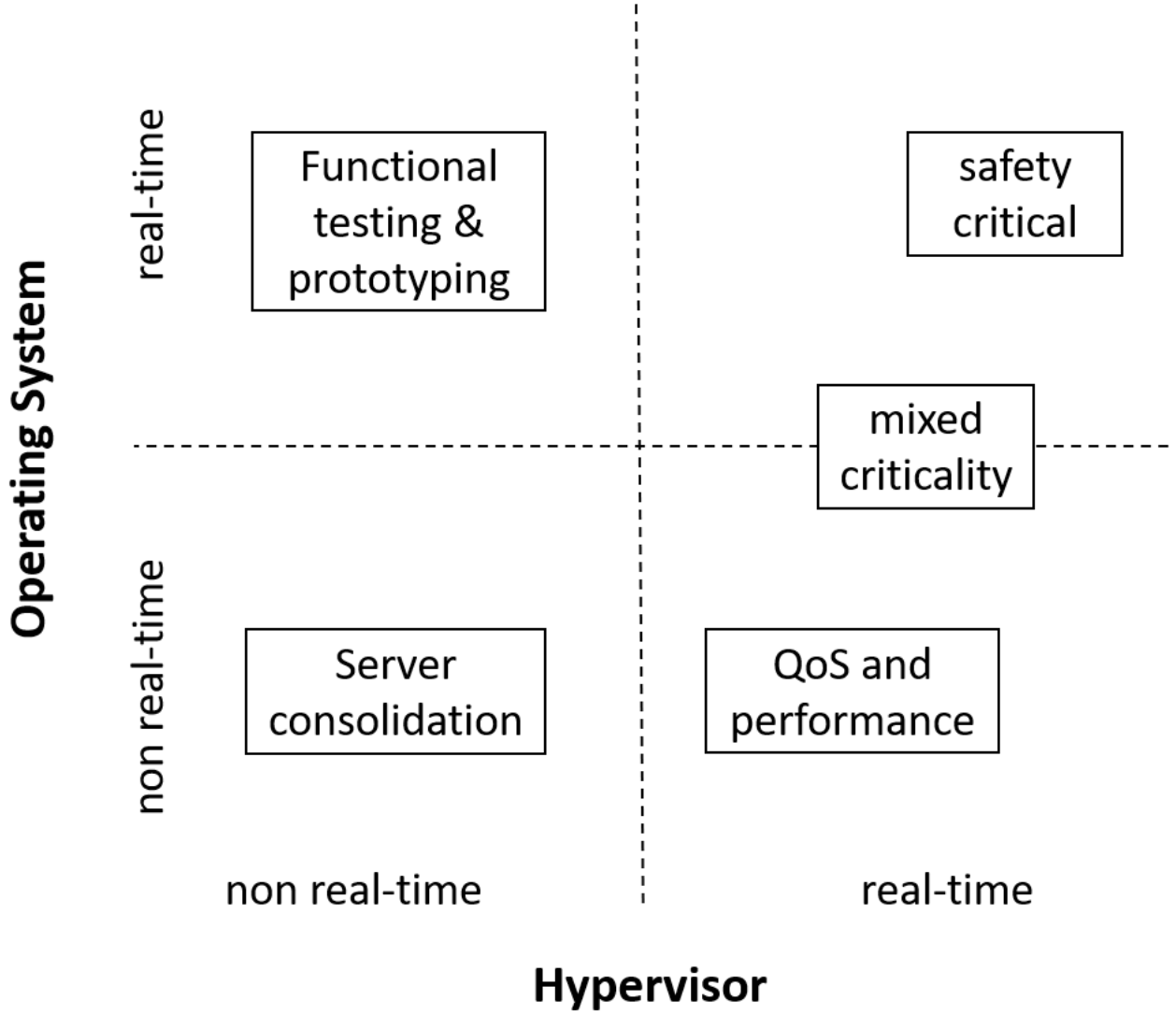
Figure: G. Kesden

Real-time hypervisors: challenges

- What about memory?
  - EPT or alike mechanisms used for spatial partitioning
  - Cache coloring and horizontal partitioning implemented at VMM level
  - And for temporal partitioning? Still open issue

- What about I/O?
  - The preferred way is static exclusive allocation of devices (pass-through)
    - Good for criticality and performance, bad for sharing and transparency
  - Some recent approaches proposed for I/O temporal partitioning, not yet available at hypervisor level

OS and hypervisor combinations



Real-time hypervisors: families

- Microkernels and separation kernels

esempio: assegno un core ad una specifica VM staticamente

  - **separation kernel:** very small type I hypervisor that utilizes hardware virtualization features to define fixed virtual machines and control information flows. They contain no device drivers, no user model, no shell access, and no dynamic memory; these tasks are all pushed up into guest software running in the VMs.
  - Examples: Jailhouse, Xtratum, VxWorks MILS, PikeOS, Nova (L4-based)
- Adaptation of general purpose hypervisors
  - Examples: XEN, KVM
- Based on advanced HW architectural features, e.g. ARM TrustZone
  - ARM TrustZone Technology features two virtual execution states (“secure” and “non-secure”) and provides time and spatial isolation between the two environments
  - Examples: LTZVisor, RTZVisor, VOSYSMonitor

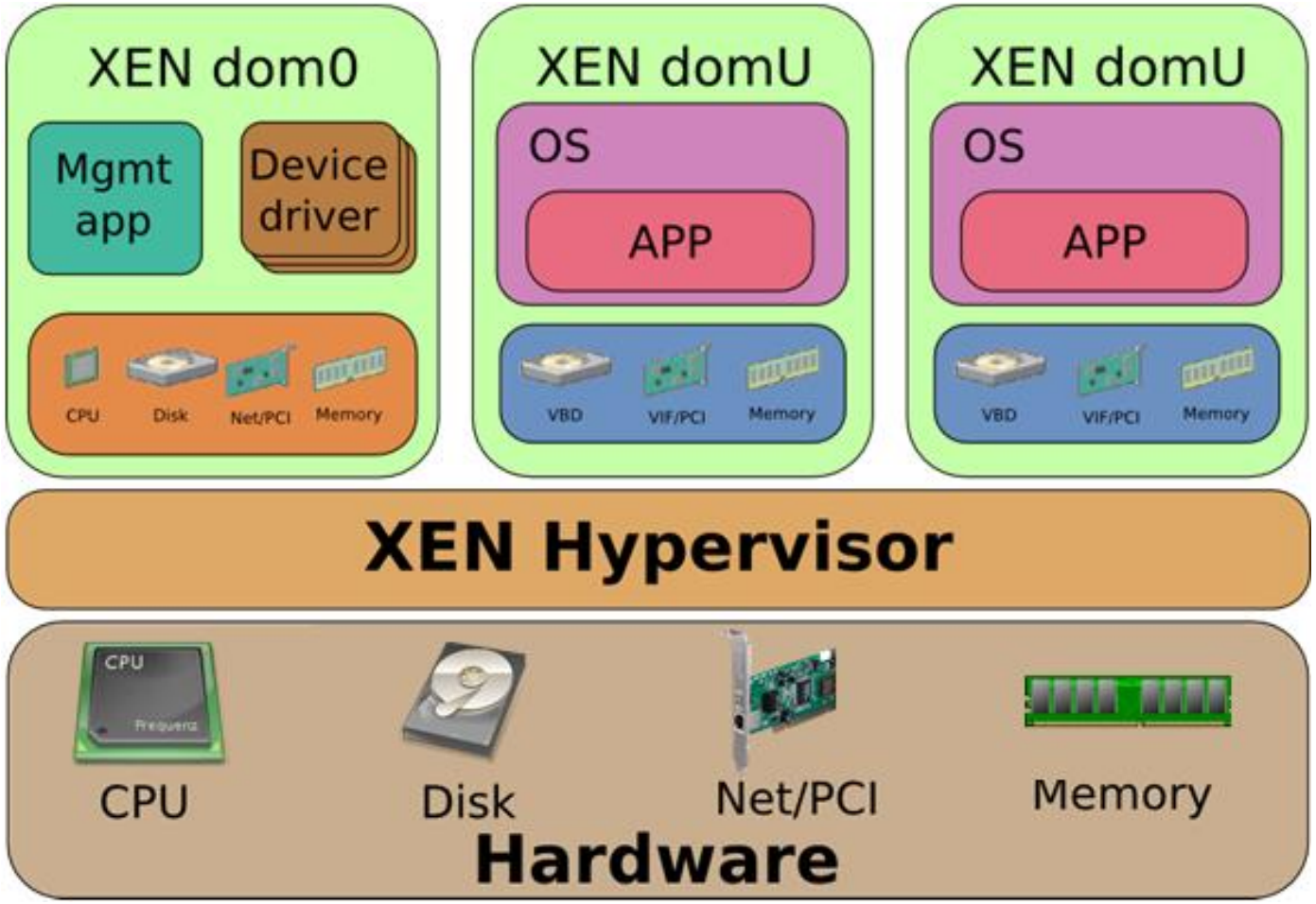
Xen

- Xen is an open-source **paravirtualization** technology that provides a platform for running multiple operating systems in parallel on one physical hardware resource
  - Today supporting full virtualization with hardware-assisted VT

- Originally developed in 2003 at the University of Cambridge Computer Laboratory

- Open source project: https://xenbits.xen.org

Xen Architecture



- **Domain0** is a privileged domain that can access the hardware resources and can manage all the other domains (e.g., create, destroy, save, restore, etc.)

- An Unprivileged Domain (***domU***) guest is more restricted.
  - Typically not allowed to perform hypercalls that directly access to the hardware.
  - Not able to manage other domains or the hypervisor configuration

- The **Xen hypervisor** is the basic abstraction layer of software that sits directly on the hardware below any operating systems.
  - It is responsible for CPU scheduling (VCPU to CPU assignment) and
  - memory partitioning

XEN virtualization mechamisms

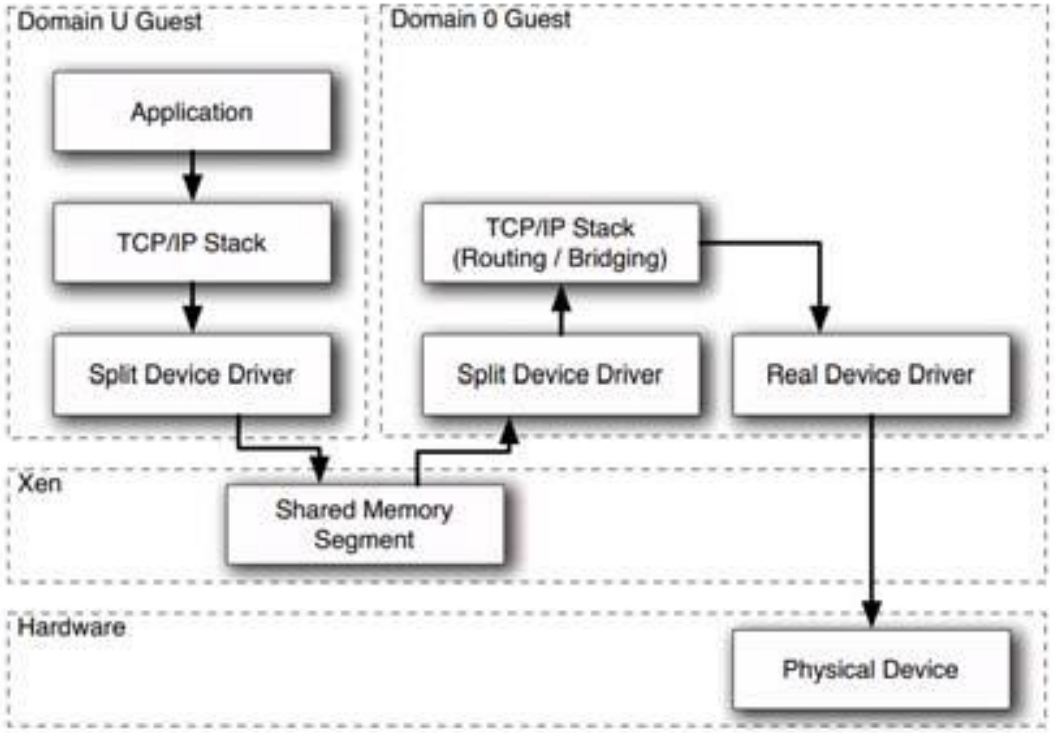
- **PV**: everything is para-virtualized
- **HVM**: full hardware emulation (through qemu) for devices (some para-virtualized devices, too); use CPU virtualization extensions (Intel VT-x, etc...)
- **PVH**: hardware virtualization for the CPU + para-virtualized devices (trade-off between the two)
  - Side Note[1]: use PV or PVH + PREEMPT_RT for real-time! ☺

Guest Kernel	PV	PVH	HVM
NRT	$661\mu s$	$1276\mu s$	$1187\mu s$
RT	$178\mu s$	$216\mu s$	$4470\mu s$

[1] L. Abeni, D. Faggioli. “An experimental analysis of XEN and KVM latencies”. At ISORC 2019

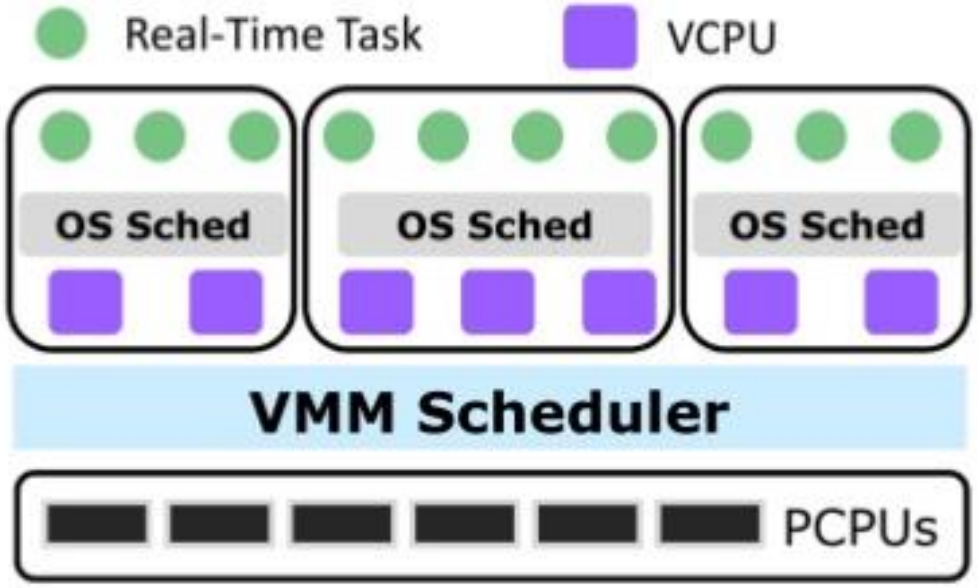
XEN I/O management

- The default is I/O paravirtualization, with full drivers available in Dom0 (and not in the VMM)
  - dom0 is a privileged domain that can access all the hardware in the system
  - The OS running on dom0 has the device drivers and performs I/O operations on behalf of domUs
  - Shared memory is used for the communication between a domU and dom0



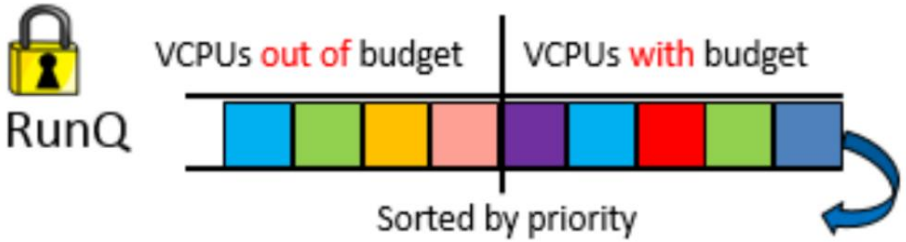
RT-XEN

- A satellite project, today included in the Xen mainline
- Born as multicore real-time hypervisor with **hierarchical scheduling**
- Schedules Virtual CPU (VCPUs) on Physical CPUs (PCPUs) using either EDF or Deadline Monotonic
- and managing the CPU budget with a Polling Server or a Deferrable Server



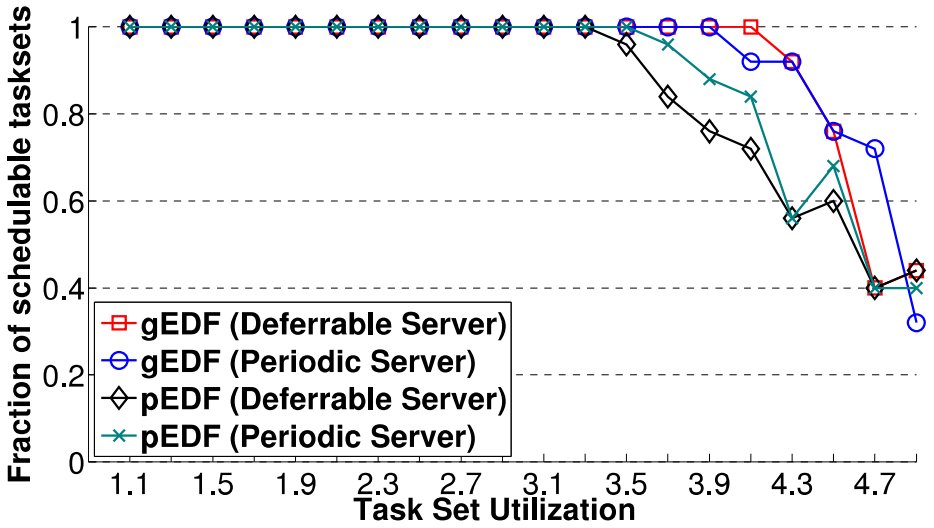
RT-XEN: runqueue

- In case of global scheduling, a single queue is managed, ordered on the basis of the priority (according to EDF or DM) and distinguishing VCPU with budget (on the head) from the ones without budget (on the tail)

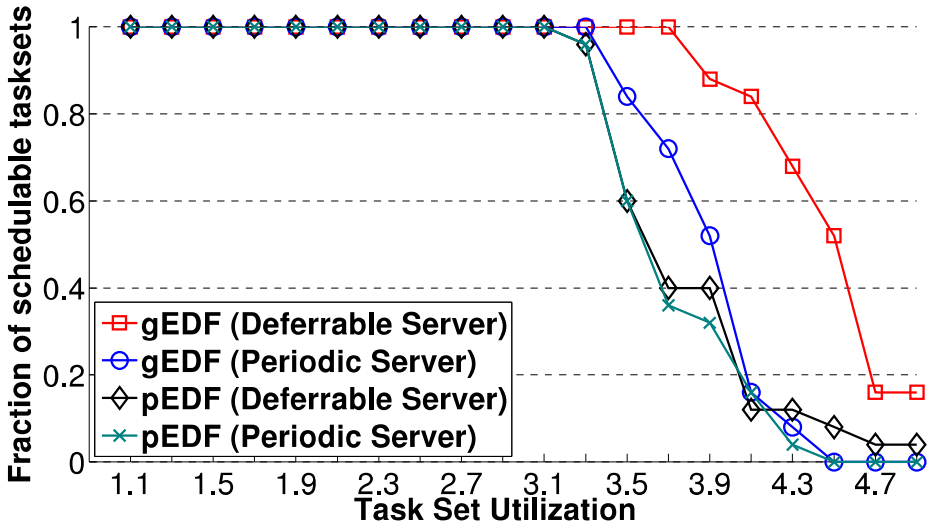


- If partitioning scheduling is used, every PCPU will have its own runqueue

RT-XEN: Selecting the best scheduler/server combination



(a) Guest OS with pEDF



(b) Guest OS with gEDF

- gEDF + Deferrable server gives the best results, and are now part of the XEN mainline (RTDS scheduler)

Xen schedulers

- Today, thanks to the RT-XEN project efforts, Xen includes several vCPU schedulers:
  - **Credit2**: proportional fair share (weighted round robin) ideal for non-real time deployments
  - **RTDS**: global EDF with reservation servers based on the Deferrable Server
  - **Arinc 653**: fixed time slices
  - **Null**: fixed pCPU to vCPU pinning

RTDS scheduler

- RTDS (*real time deferrable server*): scheduler that provides a guaranteed bandwidth share to guest VMs
  - Each vCPU has its own budget and period
  - The vCPU executes the tasks for the assigned budget each period
  - Every vCPU is replinshed with its own budget when its period starts
- Every vCPU executes as a *deferrable server*:
  - If it has a task to execute, the budget is consumed
  - The budget is preserved along all the period if no tasks have to be executed
- Follows the *Global Earliest Deadline First (gEDF)* policy for vCPU scheduling
  - The vCPU with the most imminent deadline (e.g., period of the DS) are scheduled first

https://wiki.xenproject.org/wiki/RTDS-Based-Scheduler

RTDS configuration

- The use of RTDS can be declared in the bootloader conf file (grub.cfg) or using the command line, for instance:

```
root@xen:~# xl cpupool-create name=\"pool-rt\" sched=\"rtds\" cpus=[4,5,6,8]
```

- Then VMs can be assigned to the pool
- And RTDS parameters can be changed at runtime for each VM

```
root@xen:~# xl sched-rtds -d vm1 -v all -p 500 -b 250 -e 1
```

  - This configures all VCPUs of vm1 to be scheduled with a period of 500 us and a budget of 250 us, with extra time

Other XEN's "real-time" features

- Memory:
  - Support for cache coloring
- I/O:
  - Allows device pass-trough, other than device emulation and paravirtualized I/O via dom0
    - Suitable for XEN "dom0less" deployments
  - Supports SR-IOV

Jailhouse

- Jailhouse was developed at **Siemens** in 2013
- **Static partitioning hypervisor** that runs bare metal but cooperates closely with Linux
- Real-time and/or safety tasks on **multicore platforms** (**AMP**) along GPOS
- Launches each guest with his resources set (cores, peripheral, memory)
- **Strong and clean isolation**
- **Bare-metal-like performance** and **latencies**
  - No need to modify Linux, <10k LOC/arch
- Supports x86, ARM
- Open source (GPLv2): https://github.com/siemens/jailhouse

Jailhouse Pillars

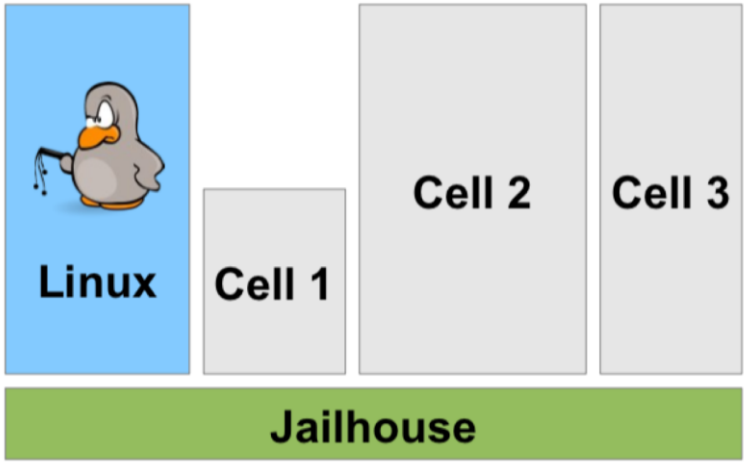
- **Use virtualization for isolation**
- **Prefer simplicity over features**
  - Resource access control instead of resource virtualization
  - 1:1 resource assignment instead of scheduling
  - Partition booted system instead of booting Linux
- **Offload work to Linux**
  - System boot
  - Jailhouse and partition (VM) loading & starting
  - Control and monitoring

Jailhouse concepts

- Jailhouse **doesn't emulate resources you don't have**
- Splits physical hardware into **isolated compartments**
  - **Cells** that are fully dedicated to guest OS and software programs called **inmates**
- **Two kind of cells**
  - **Root cell**: runs the ‘’host’’ Linux OS
  - **Other cells:** borrow CPUs and devices from the root cell as they are created

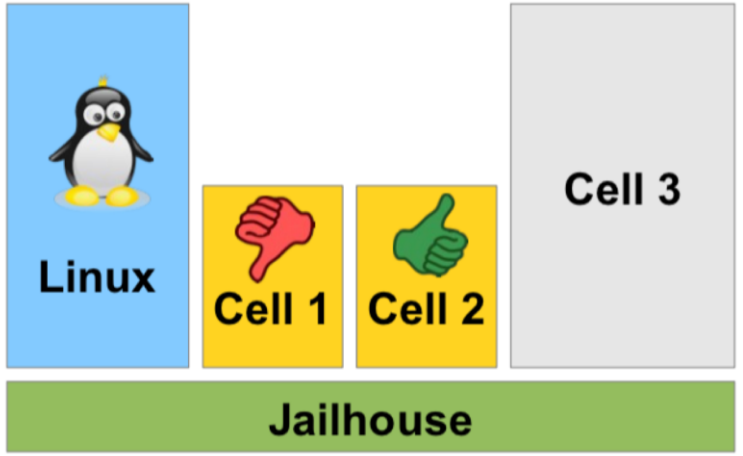
Jailhouse modes

Open Model



- **Linux (root cell) is in control**
- Cells not involved in management decisions
- Sufficient if root cell is trusted

Safety Model



- **Linux controls, but...**
- Cells can be configured to vote over management decisions
- Building block for safe operation