# CS 477:

# Advanced Operating Systems

## Virtualization

# This week

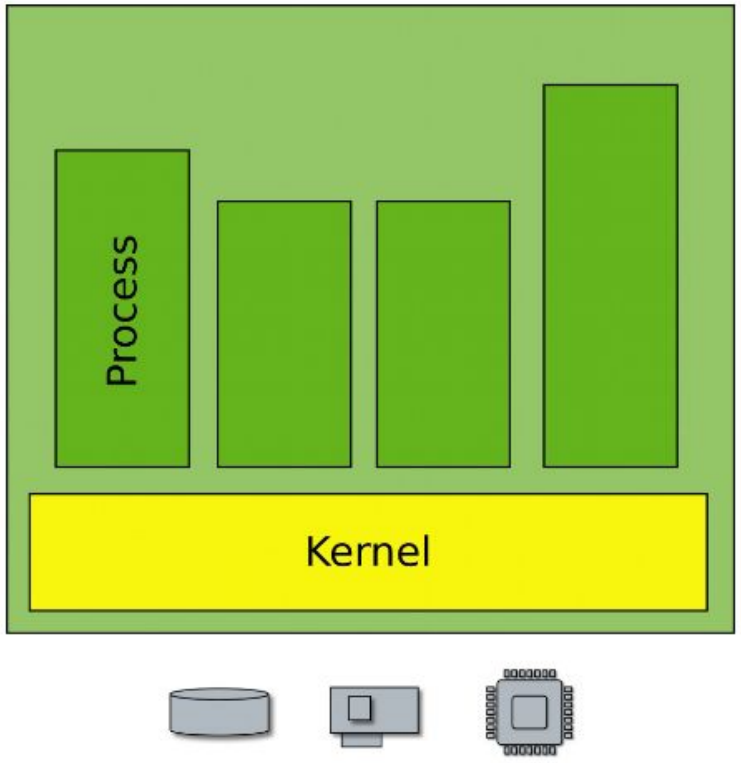
- Virtualization
  - CPU
  - Memory
  - IO
- Paper: My VM is Lighter (and safer) than your Container

# What is virtualization?

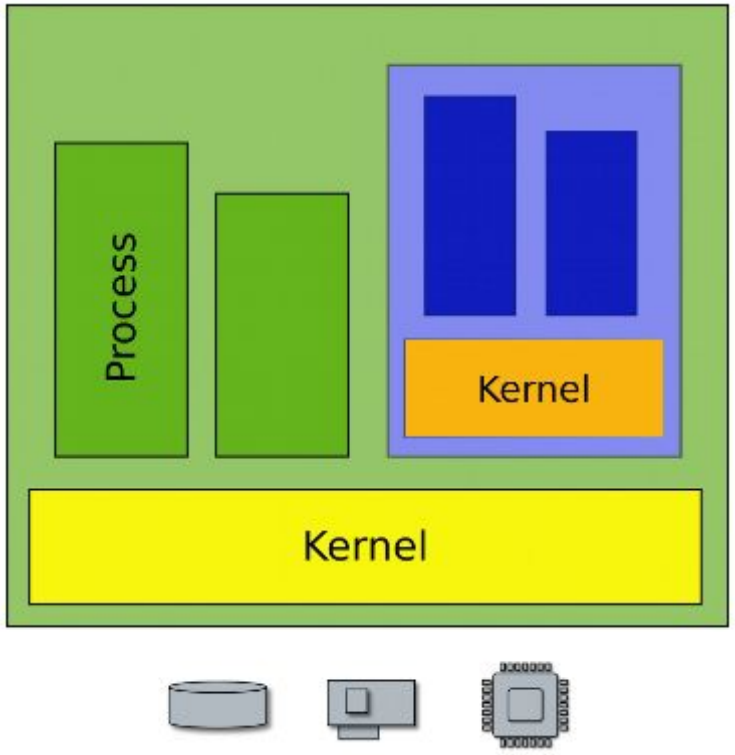
Process of presenting and partitioning computing resources in a **logical way** rather than what is dictated by their **physical reality**

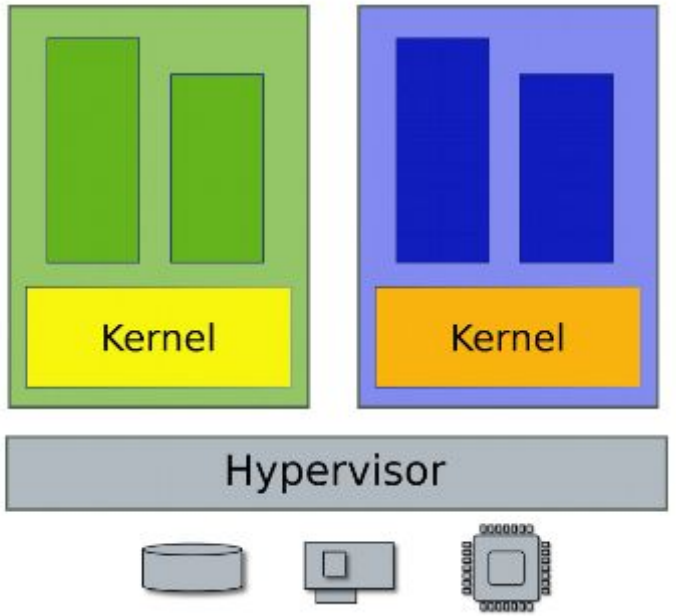
- Extend or replace an existing interface to mimic the behavior of another system
- For portability, flexibility …

# Traditional OS vs virtual machines (VMs)




Traditional OS



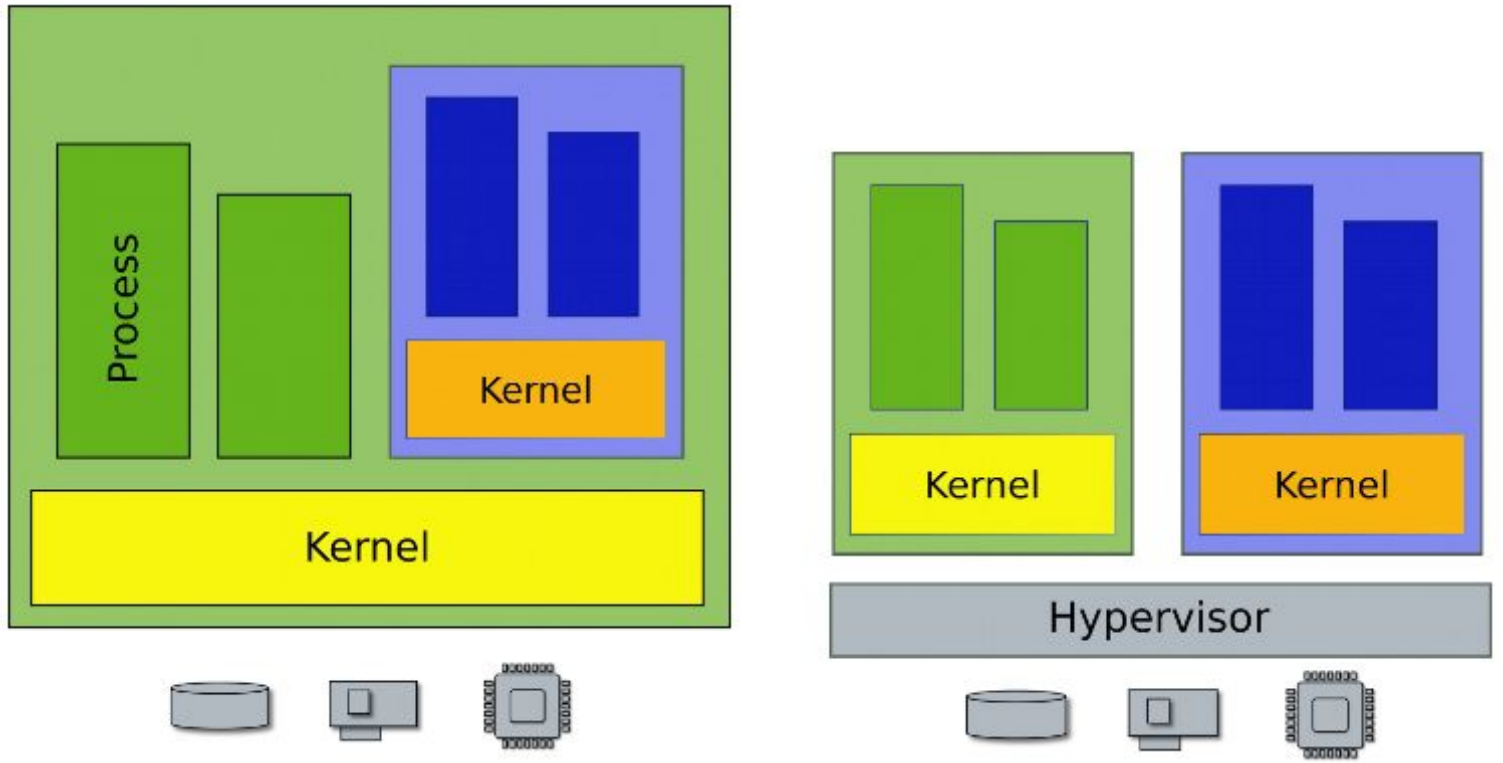




Virtual machine

# Traditional OS vs virtual machines (VMs)

Virtual machine: An execution environment identical to a physical machine, with the ability to execute a full OS

*Process: OS :: OS: Virtualization layer* (Virtual machine monitor, VMM, or hypervisor)




Virtual machine

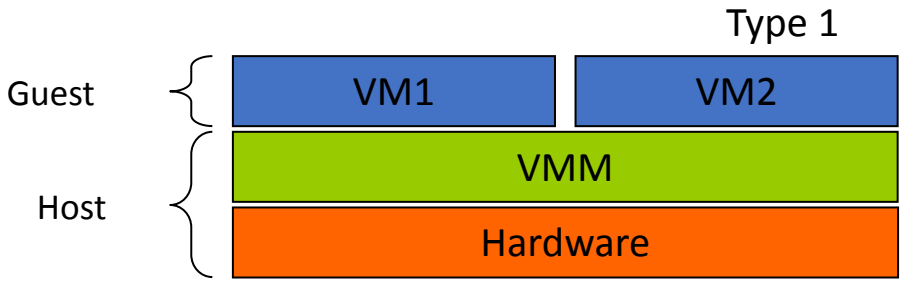
# History

- Virtual machines were popular in 60s-70s
  - Share resources of mainframe computers [Goldberg 1974]
  - Run multiple single-user OSes
- Interest lost in 80s-90s’
  - Development of multi-user OS
  - Rapid drop in hardware cost

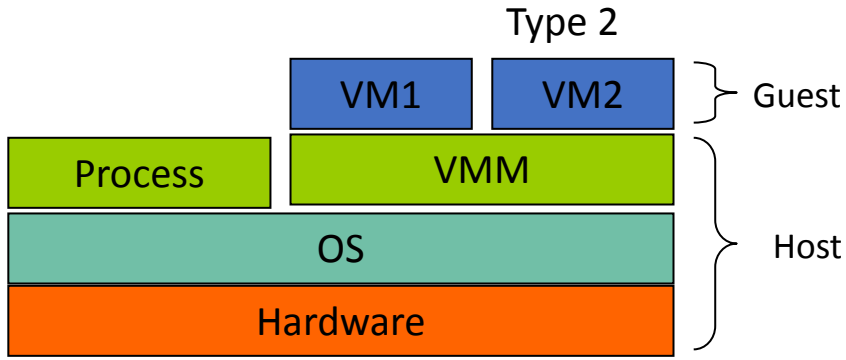
# Virtualization

- What is virtualization?
  - Virtualization is a way to run multiple operating systems and user applications on the same hardware
  - Virtual Machine Monitor (Hypervisor) is a software layer that allows several virtual machines to run on a physical machine

- Types of VMMs
  - Type-1: hypervisor runs directly on hardware
  - Type-2: hypervisor runs on a host OS




Xen, VMware ESX, Microsoft Hyper-V




KVM, VMware Workstation, Sun VirtualBox, QEMU

- Virtual Machine Monitor (VMM) = Hypervisor = Host OS
- Virtual Machine (VM) = Guest OS

# Virtualization advantages

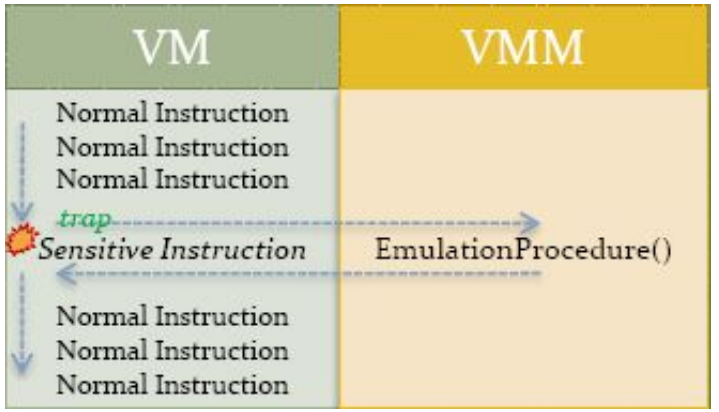
- Isolation
  - Limits security exposures
  - Reduces spread of risks
- Roll-Back
  - Quickly recovers from security breaches
- Abstraction
  - Limits direct access to hardware
- Portability
  - Disaster recovery
  - Switches to “standby” VMs
- Deployment
  - Distributes workloads
  - Customizes guest OS security settings

# Virtualization types

- Applications
  - Server virtualization
    - Green IT
    - Xen, VMware ESX Server
  - Desktop virtualization
    - VMware, VirtualBox, Citrix's Xen HDX
  - Mobile virtualization
    - Secure execution
    - Xen on ARM
  - Cloud computing
    - Storage/platform cloud services
    - Amazon EC2, MS Azure, Google AppEngine
  - Emulation
    - iPhone/Android emulator
    - Qemu, Bochs

# Processor virtualization

- Classic virtualization
  - Trap & Emulate
    - For an architecture to be virtualizable, all sensitive instructions must be handled by VMM




  - Sensitive instructions include
    - Instruction that changes processor mode
    - Instruction that accesses hardware directly
    - Instruction whose behavior is different in user/kernel mode

# x86 is not virtualizable

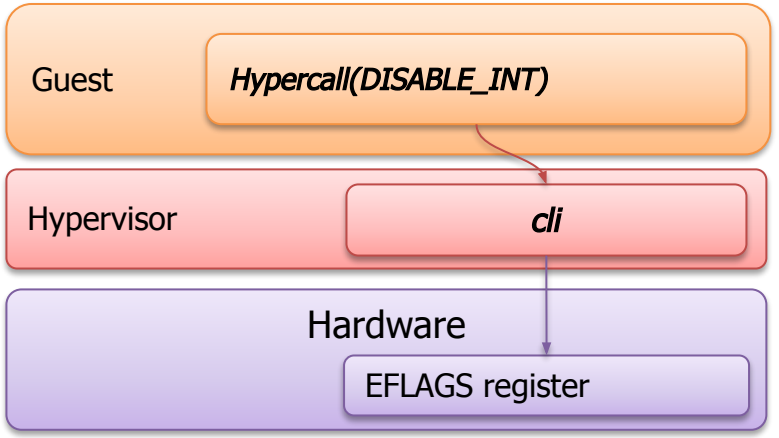
- Some instructions (**sensitive**) read or update the state of VM and don't trap (**non-privileged**)
  - 17 sensitive, non-privileged instructions

| Group | Instructions |
|---|---|
| Access to interrupt flag | pushf, popf, iret |
| Segment manipulation instructions | pop <seg>, push <seg>, mov <seg> |
| Read-only access to privileged instructions | sgdt, sldt, sidt, smsw |
| Interrupt and gate instructions | fcall, longjump, retfar, str, int <n> |

- popf doesn't update interrupt flag (IF)
  - Impossible to detect when guest disables interrupts
- push %cs can read code segment selector (%cs) and learn its CPL
  - Guest gets confused

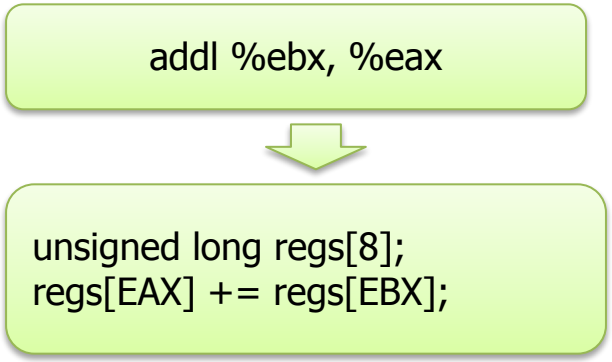
# Processor virtualization (cont'd)

- Para-virtualization
  - Requires modifications to the guest OS
    - Guest is aware that it is running on a VM
  - Example
    - Instead of doing "cli" (turn off interrupts), guest OS should do hypercall(DISABLE_INT)
  - Pros
    - Near-native performance
    - No hardware support required
  - Cons
    - Requires specifically modified guest
  - Solutions : Xen



# Processor virtualization (cont'd)

- Full-virtualization
  - Emulation
    - Process of implementing the interface and functionality of one system on a different system
    - Do whatever the CPU does, but in software
    - CPU emulation
      - Fetches and decodes the next instruction
      - Executes using the emulated registers and memory
    - Pros
      - No hardware support required
      - Simple
    - Cons
      - Very slow
    - Solutions : Bochs



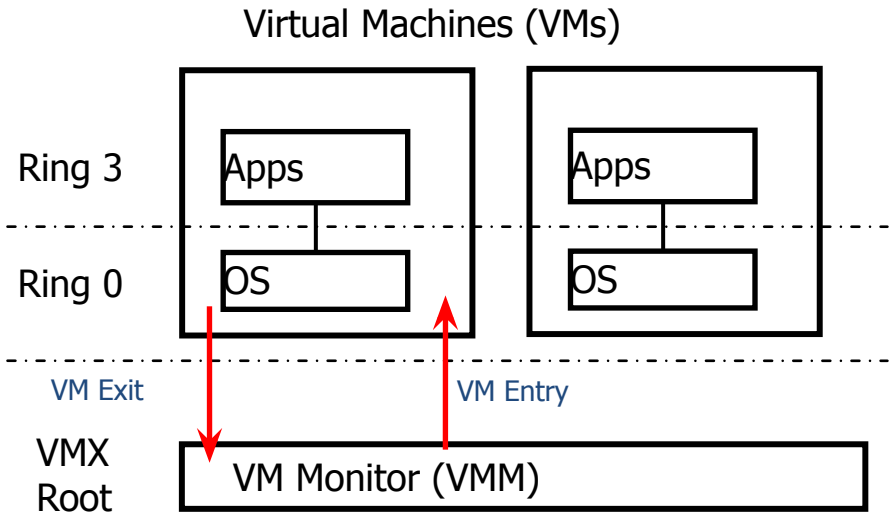
# Processor virtualization (cont'd)

- Full-virtualization
  - Binary translation
    - Translates code block to safe code block (like JIT) directly
    - Dynamically translates privileged instructions to normal instructions which can be executed in user mode
    - Pros
      - No hardware support required
      - Fast
    - Cons
      - Hard to implement
        - » VMM needs x86-to-x86 binary compiler
    - Solutions : VMware, QEMU

JIT : just in time compilation

# Processor virtualization (cont'd)

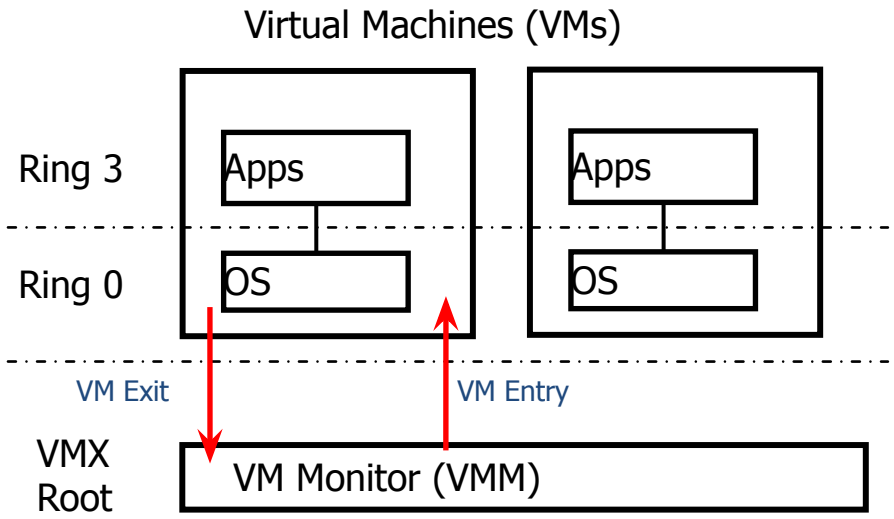
- Full-virtualization
  - Hardware-assisted virtualization
    - Runs the VM directly on the CPU
      - No emulation
    - Integrates new execution mode into the CPU by extending the instruction set and control structure
    - Pros
      - Fast
    - Cons
      - Need hardware support
        - » AMD SVM
        - » Intel VT
    - Solutions : KVM, Xen




AMD SVM : Secure Virtual Machine
Intel VT : Virtualization Technology

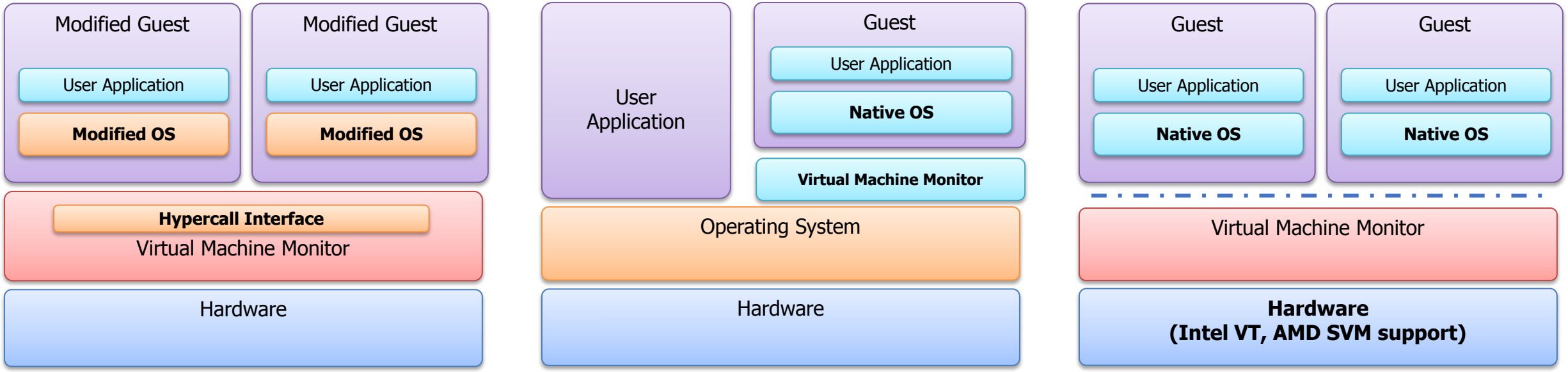
# Intel VMX

- VMX(Virtual Machine Extension) supports virtualization of processor hardware.
- Two new VT-x operating modes
  - Less-privileged mode (VMX non-root) for guest OSes
  - More-privileged mode (VMX root) for VMM
- Two new transitions
  - VM entry to non-root operation
  - VM exit to root operation
- Execution controls determine when exits occur
  - Access to privilege state, occurrence of exceptions, etc.
  - Flexibility provided to minimize unwanted exits
- VM Control Structure (VMCS) controls VMX operation
  - Also holds guest and host state



# Processor virtualization (cont'd)

- Comparison




Para-virtualization

Full-virtualization
(Emulation, Binary Translation)

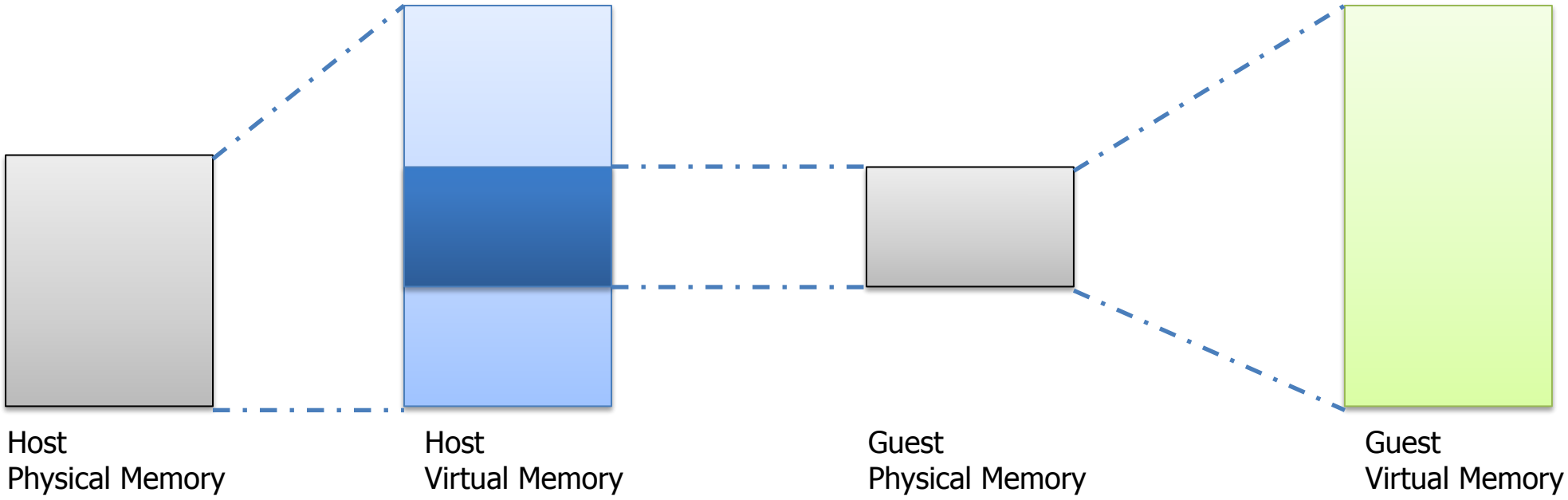
Full-virtualization
(Hardware-assisted VT)

# Processor virtualization (cont'd)

- Comparison

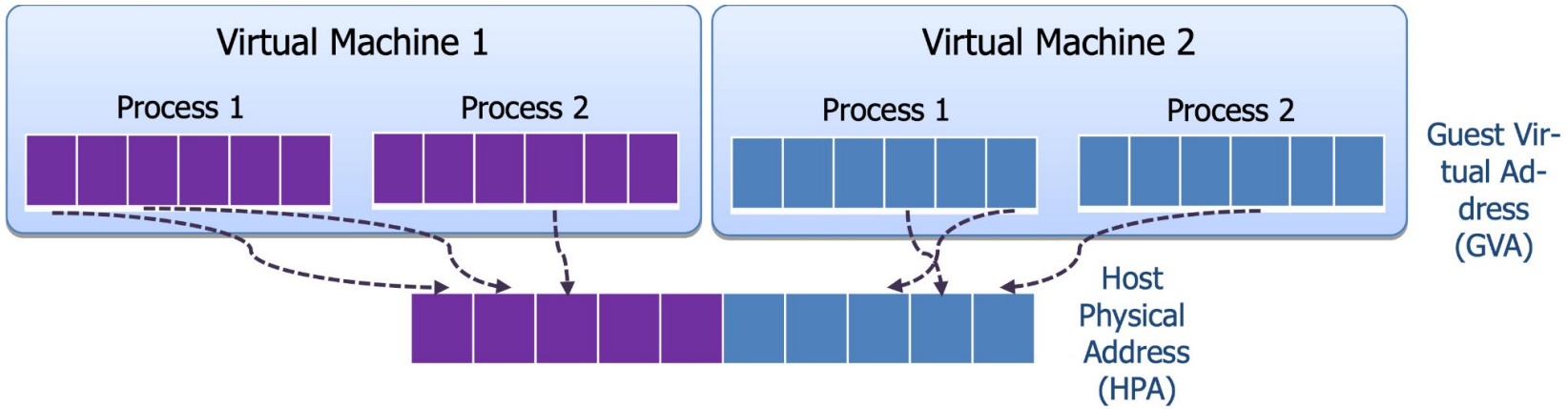
| | Para-virtualization | Full-virtualization (Emulation) | Full-virtualization (Binary translation) | Full-virtualization (Hardware-assisted VT) |
|---|---|---|---|---|
| Speed | Very Fast (Almost Native) | Very Slow | Fast | Fast |
| Guest Kernel Modification | Yes | No | No | No |
| Support Other Arch | No | Yes | No | No |
| Solutions | Xen, VMWare ESX | Bochs | VMWare, QEMU | KVM, Xen |
| Purposes | Server virtualization | Emulator | Desktop virtualization | Desktop virtualization |

# VM memory map



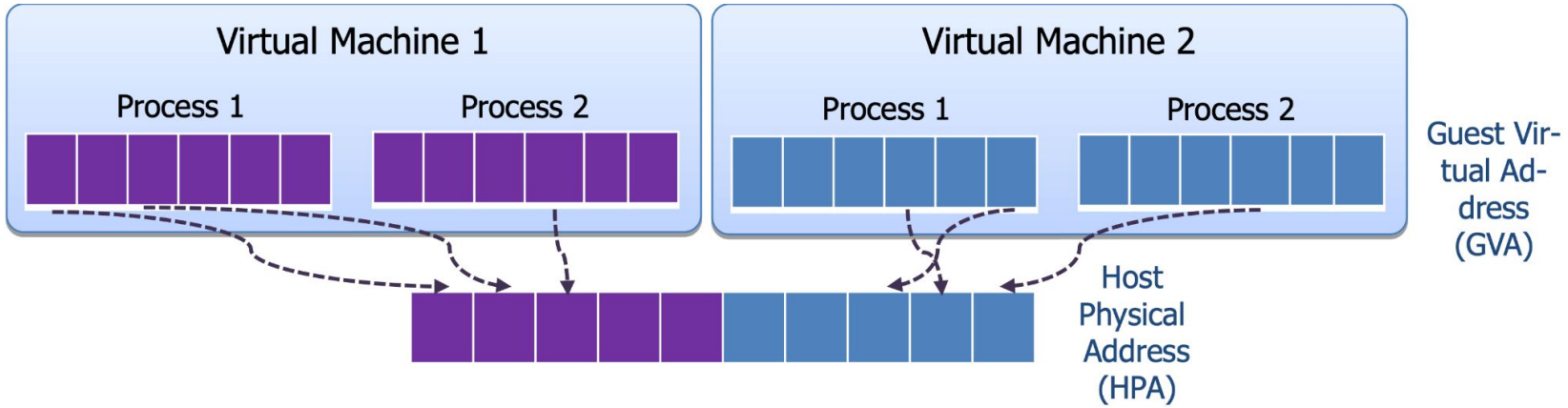
# Memory virtualization

- Direct Paging
  - Guest OS directly maintains a mapping of Guest Virtual Address to Host Physical Address (GVA 📫 🕹 HPA).
  - When a logical address is access, the hardware walks these page tables to determine the corresponding physical address.
  - Dedicated physical memory region is allocated at the initialization of guest OS.
  - Pros
    - Simple to implement
    - High performance (no virtualization overhead)
  - Cons
    - Need to modify guest kernel (not applicable to closed-source OS)
    - Inflexible memory management

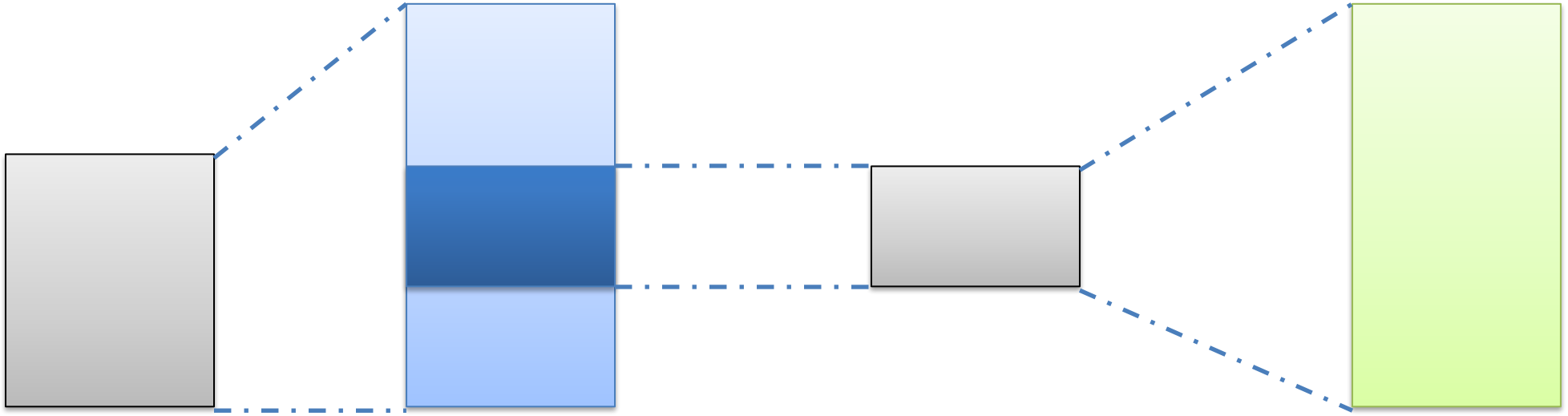


# Memory virtualization (cont'd)

- Shadow Paging
  - VMM maintains GVA📪☮GPA mappings in its internal data structures and stores GVA📪☮HPA mappings in shadow page tables that are exposed to the hardware.
  - The VMM keeps these shadow page tables synchronized to the guest page tables.
  - This synchronization introduces virtualization overhead when the guest updates its page tables.
  - Pros
    - Support unmodified guest OS
  - Cons
    - Hard to implement and maintain
    - Large virtualization overhead



# Shadow paging



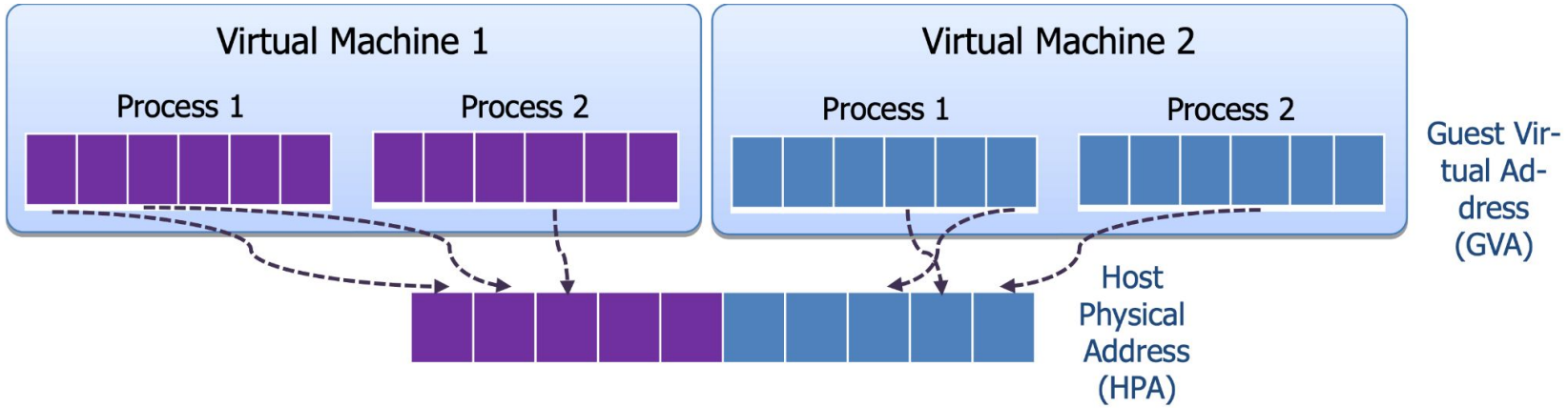

- MMU with host page table created by host kernel
- MMU with shadow page table created by hypervisor

- One-to-one mapping

- Guest page table
- It represents some ‘logical’ address in guest environment. It cannot be used for MMU, because guest physical address is another virtual address.

# Memory virtualization

- Nested Paging
  - Guest operating system continues to maintain GVA📪📪☮GPA mappings in the guest page tables.
  - But the VMM maintains GPA📪☮HPA mappings in an additional level of page tables, called nested page tables.
  - Both the guest page tables and the nested page tables are exposed to the hardware.
  - When a logical address is accessed, the hardware walks the guest page tables as in the case of native execution, but for every GPA accessed during the guest page table walk, the hardware also walks the nested page tables to determine the corresponding HPA.
  - Pros
    - Simple to implement
    - Support unmodified guest OS
  - Cons
    - H/W supports is needed.
    - Larger TLB footprint

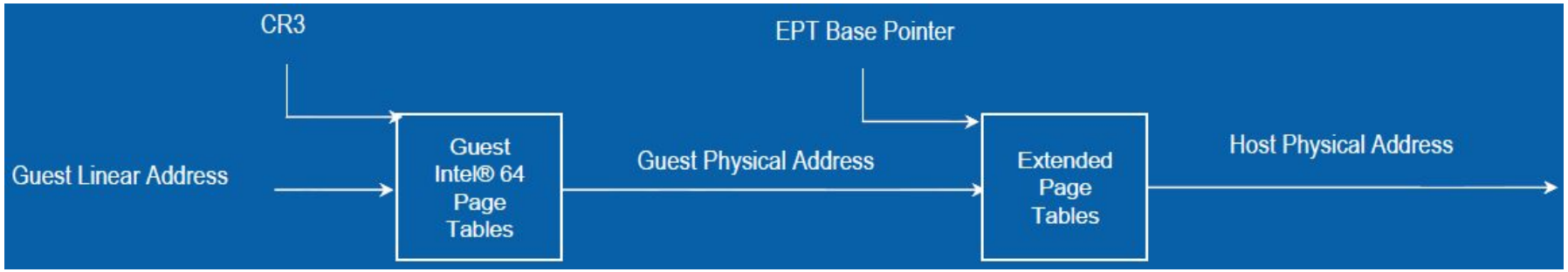


# Memory virtualization

- Comparison

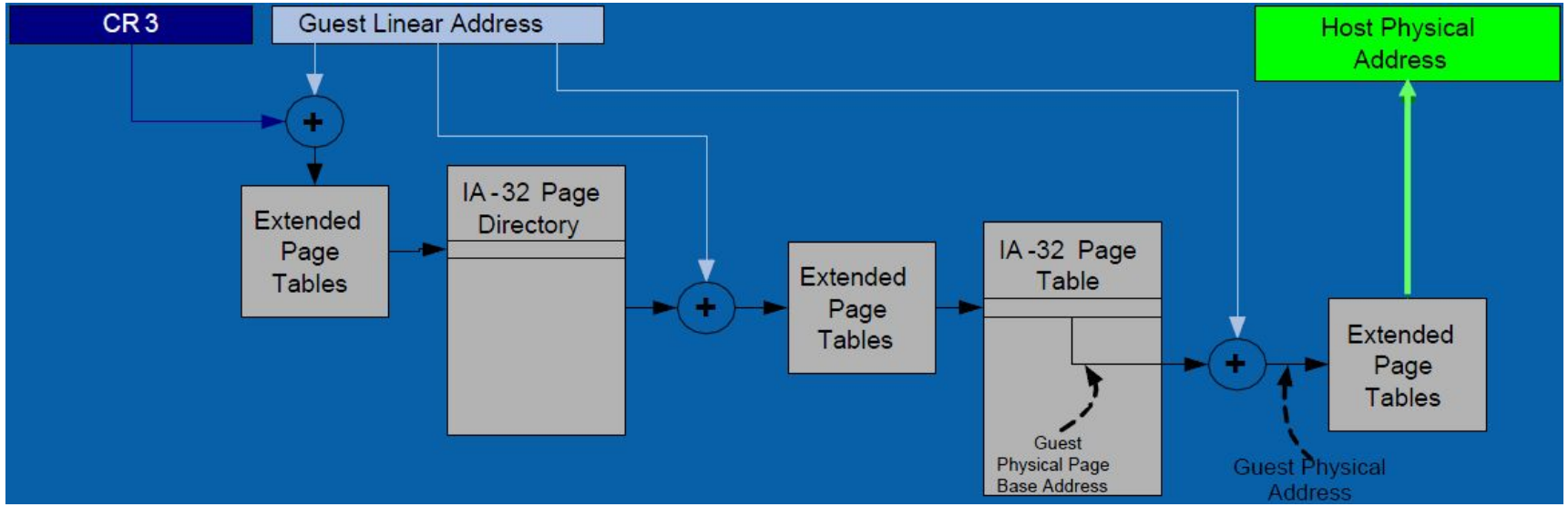
| | Direct Paging | Shadow Paging | Nested Paging |
|---|---|---|---|
| **Speed** | **Very Fast (Almost Native)** | **Very Slow** | **Fast** |
| **Guest Kernel Modification** | **Yes** | **No** | **No** |
| **Need H/W Support** | **No** | **No** | **Yes** |
| **Complexity** | **Simple** | **Complex** | **Very Simple** |

# Extended page tables (EPT)




- Feature supporting the virtualization of physical memory
- Guest-physical addresses are translated by traversing a set of EPT paging structures to produce physical addresses that are used to access memory
- Guest can have full control over page tables/events
  - CR3, CR0, CR4 paging bits, INVLPG, page fault
- VMM controls Extended Page Tables
- CPU uses both tables, guest paging structure and EPT paging structure
- EPT activated on VM entry
  - When EPT active, EPT base pointer (loaded on VM entry from VMCS) points to extended page tables
- EPT deactivated on VM exit

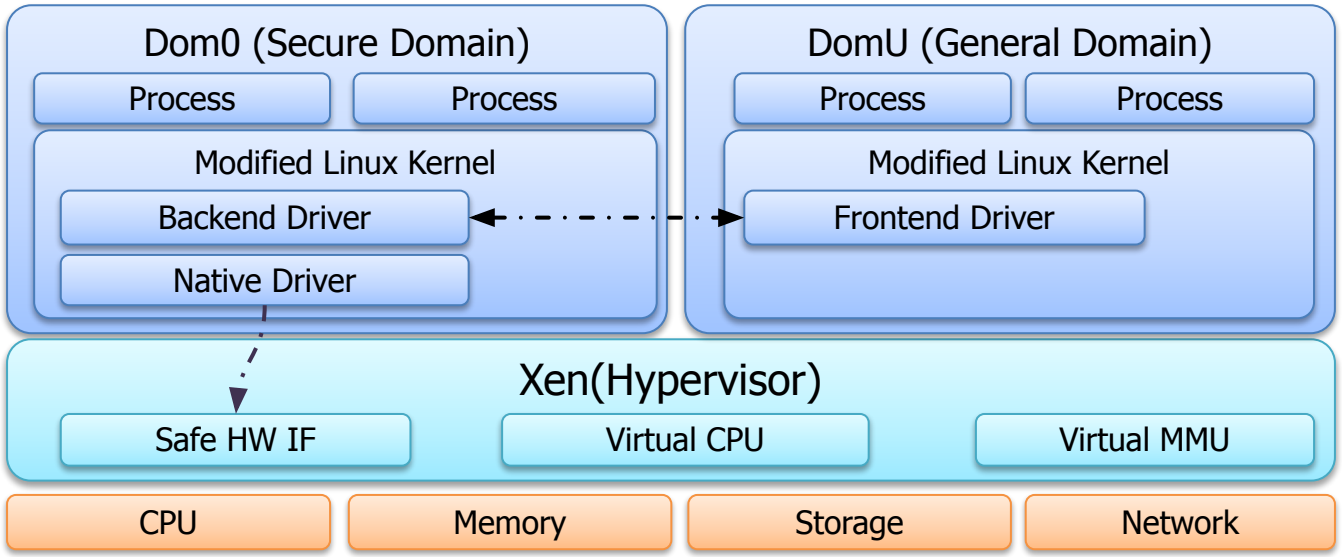
# EPT translation




- All guest-physical addresses go through extended page tables
  - Includes address in CR3, address in PDE, address in PTE, etc.
- In addition to translating a guest-physical address to a physical address, EPT specifies the privileges that software is allowed when accessing the address. Attempts at disallowed accesses are called EPT violations and cause VM exits.

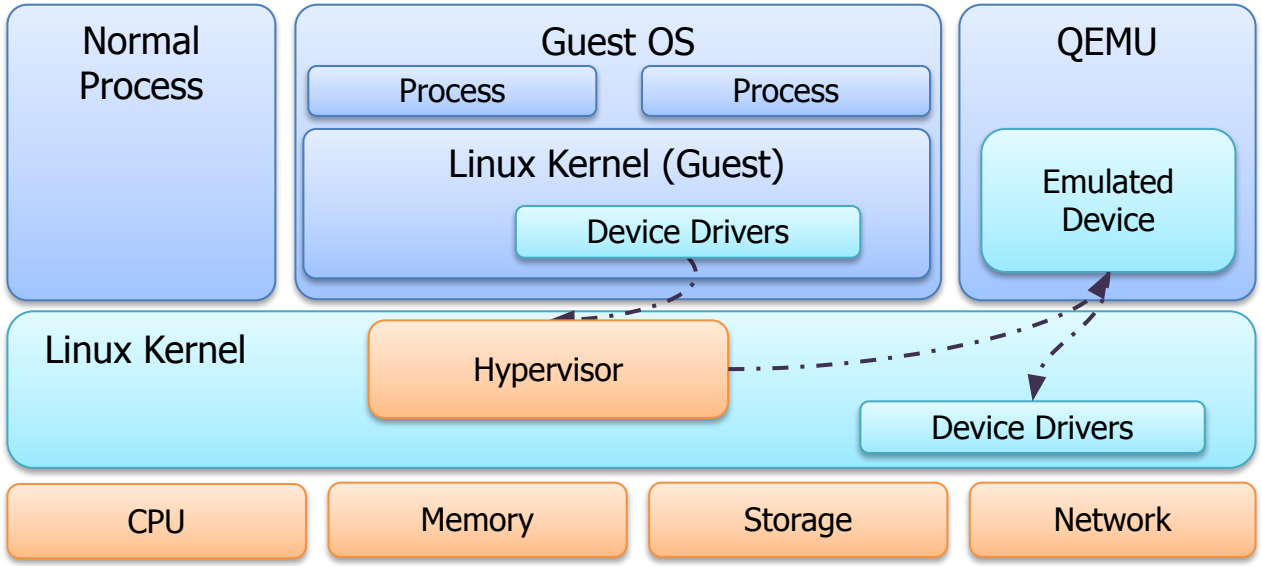
# IO virtualization

- Front-end/Back-end driver model
  - Guest OS uses para-virtualized front-end driver to send requests to backend driver
  - Back-end driver on secure domain receives the requests, performs actual IO using the native driver



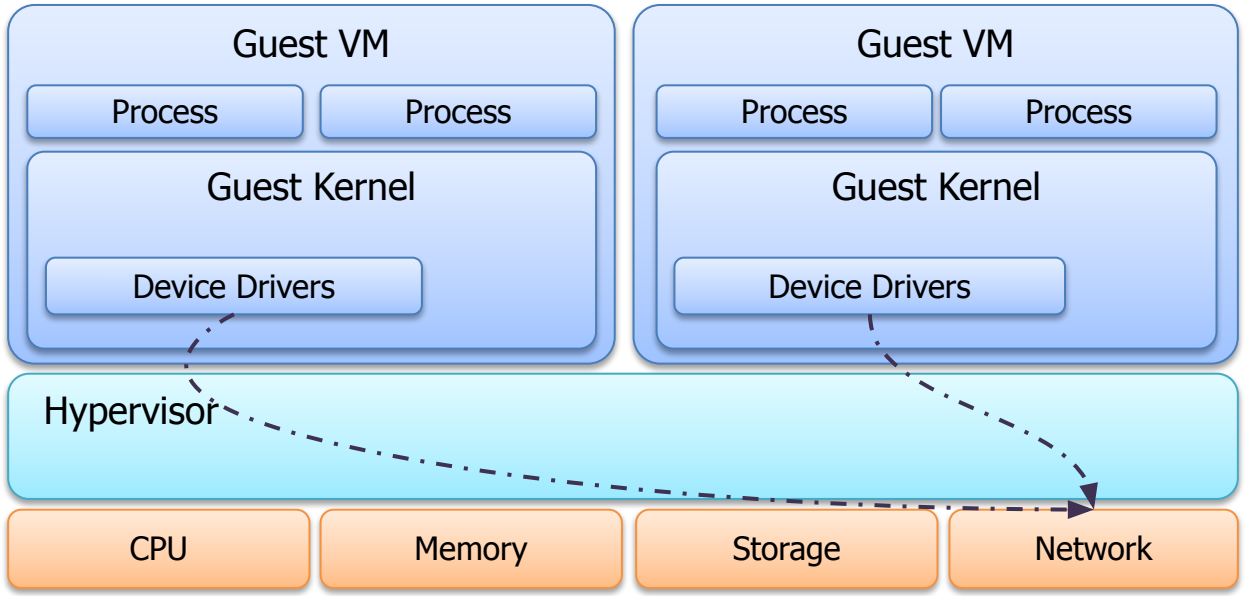
# IO virtualization (cont'd)

- Emulation
  - Behavior of a particular device is emulated as a software module.
  - Guest OS uses the native device driver for the particular device.
  - VMM intercepts all the access from guest OS to the device.
  - The intercepted accesses are sent to the emulated device.
  - The Emulated device do the actual IO operations.

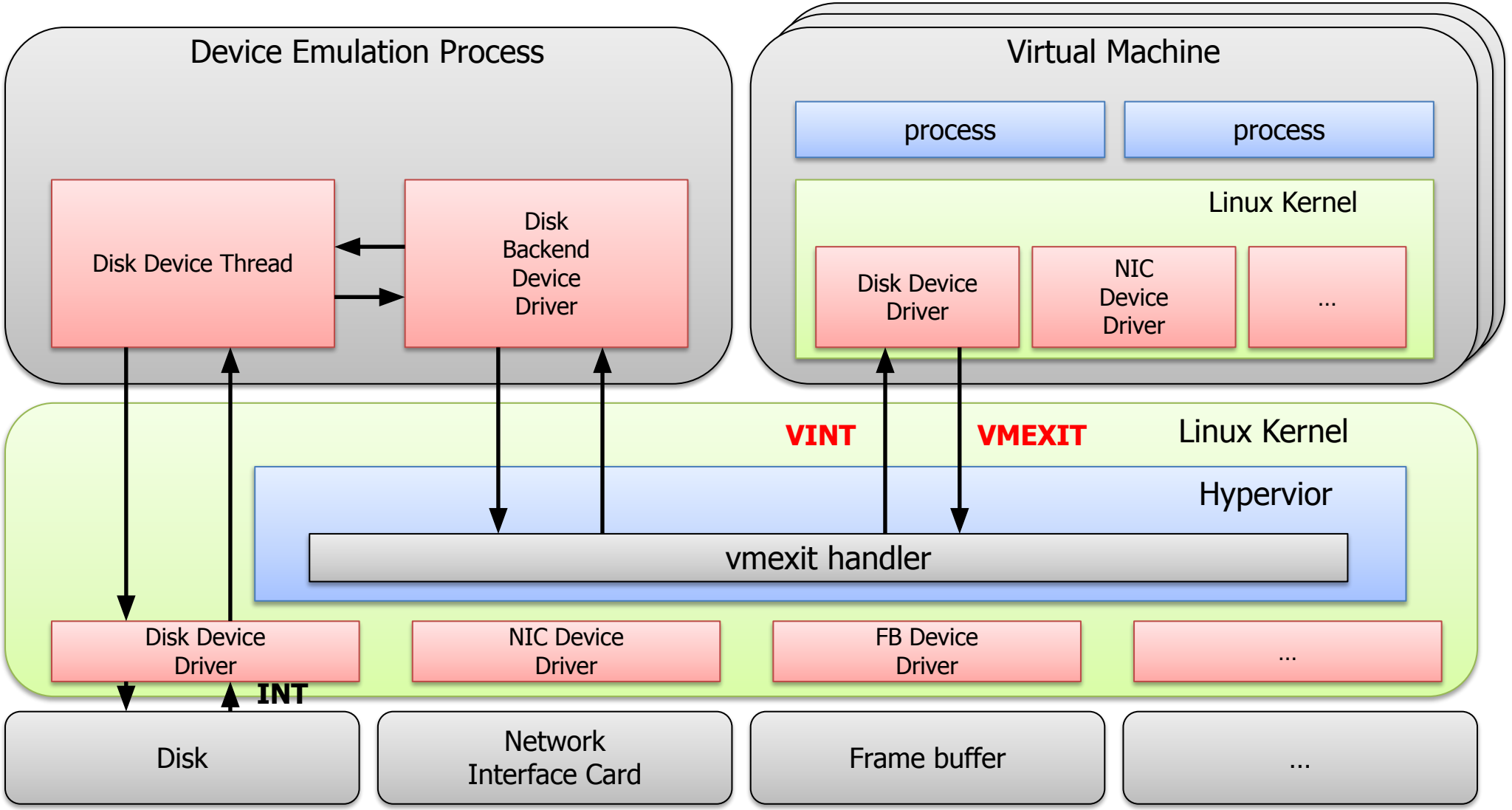


# IO virtualization (cont'd)

- H/W Assisted IO Virtualization
  - A specially designed H/W supports concurrent accesses from multiple guest OS.
  - Guest OS use the unmodified device driver.
  - Guest OS can access arbitrary host physical memory through DMA.
    - Intel VT-d controls the host physical memory access from the guest OS through DMA.



# IO virt. model: frontend/backend



# IO virtualization (cont'd)

- Comparison

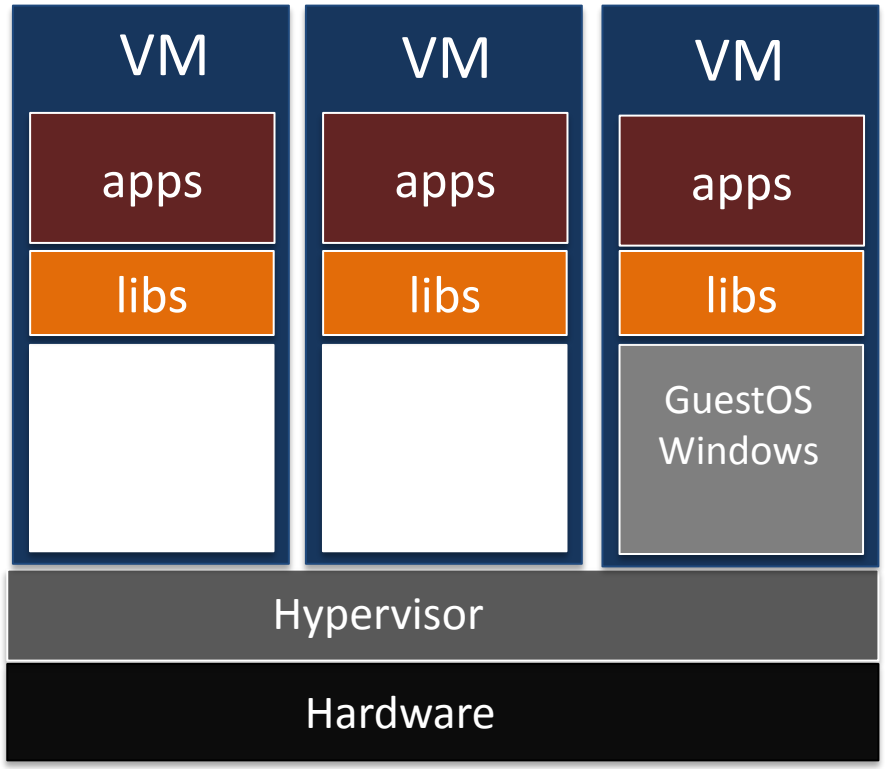
| | Front-end/Back-end Driver Model | Emulation | H/W Assisted IO Virtualization |
|---|---|---|---|
| **Speed** | Very Slow | Very Slow | Fast |
| **Device Driver Modification** | Yes | No | No |
| **Need H/W Support** | No | No | Yes |
| **Complexity** | Simple | Complex | Simple |

# This week

- Virtualization
  - CPU
  - Memory
  - IO
- **Paper: My VM is Lighter (and safer) than your Container**

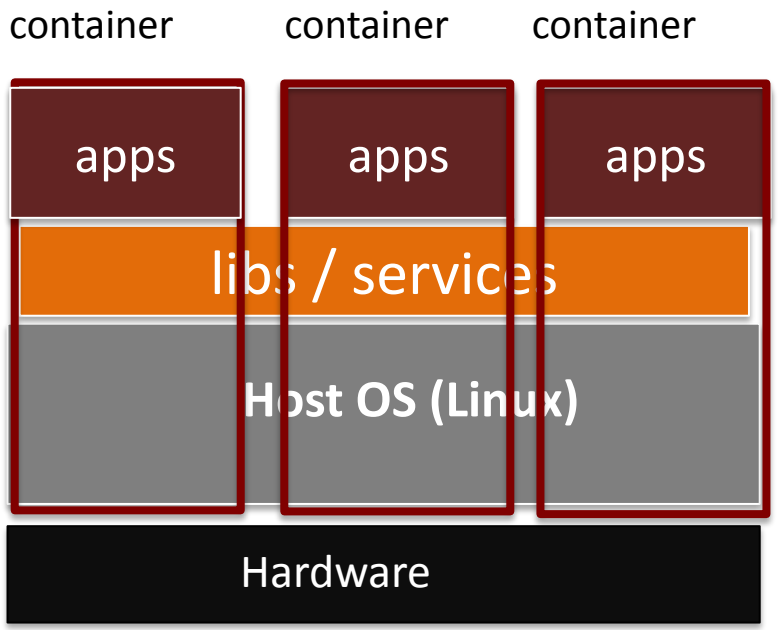
# Isolating workloads …

Virtualization




vs.

Containers

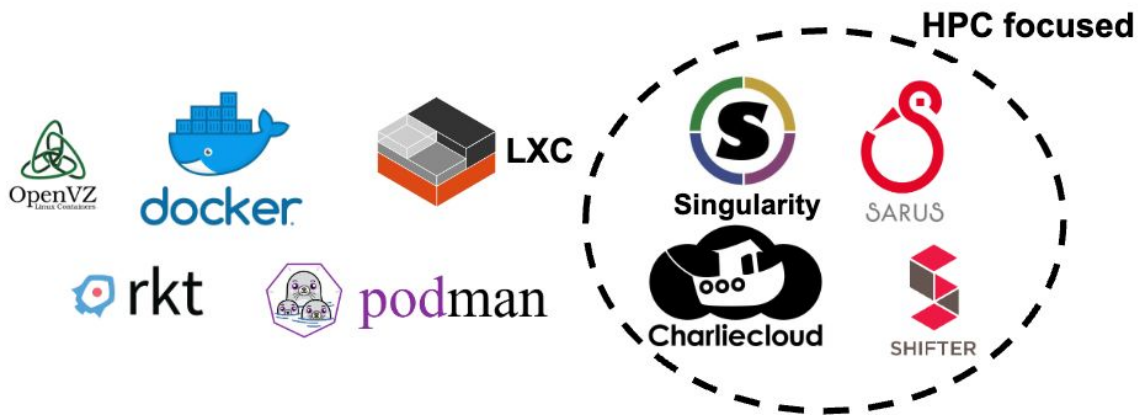


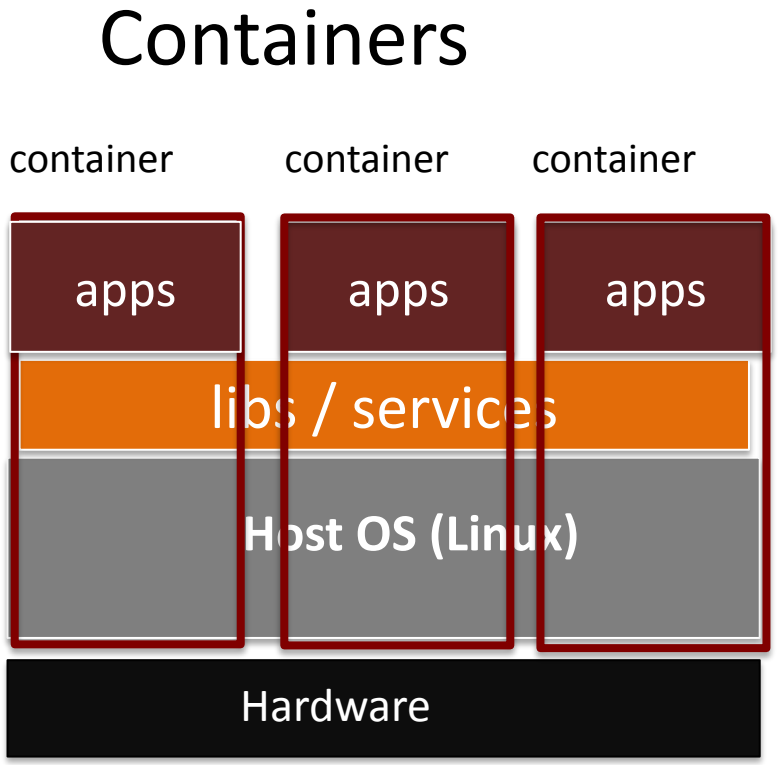

- Strong isolation
- Heavy weight

- Lightweight
- Iffy isolation

# Containers …

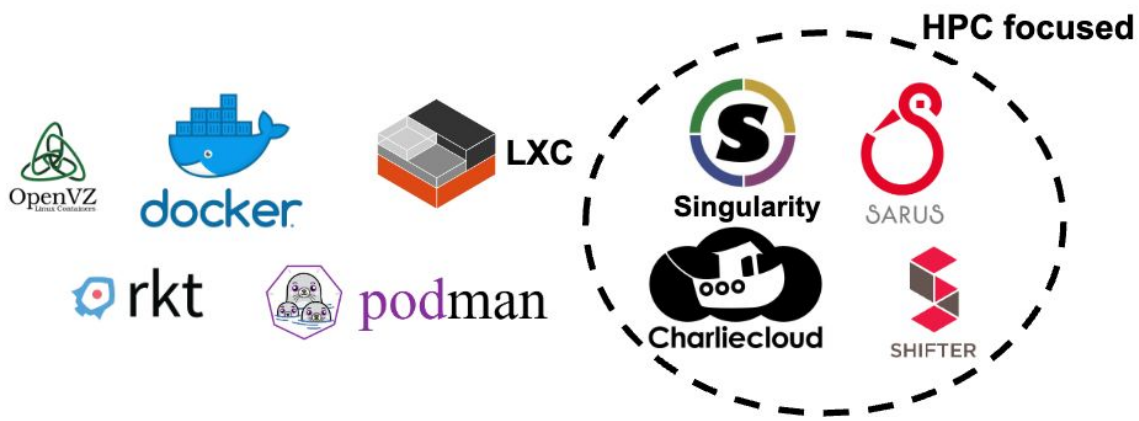
- Isolated environments to run applications / services
- Images include all software dependencies
- Prescriptive, portable, easy to build, quick to deploy

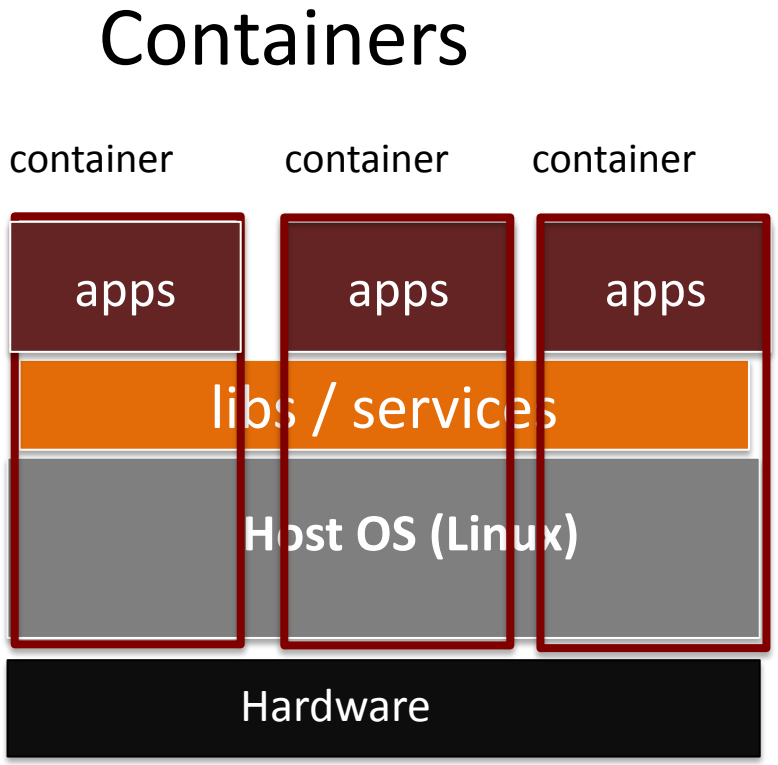


# Containers …

- Rely on abstraction feature: namespace that the kernel provides
- Popular example: docker
  - Creates containers from recipe-like files
  - Cloud-based image distribution strategy



# Issues with containers

- Containers use syscall API instead of simple x86 ABI offered by VMs
  - Difficult to secure syscalls even after several isolation mechanisms
- Have a lot of increasing number of exploits
- Monopolize or exhaust system resources

# Issues with containers

- Containers use syscall API instead of simple x86 ABI offered by VMs
  - Difficult to secure syscalls even after several isolation mechanisms
- Have a lot of increasing number of exploits
- Monopolize or exhaust system resources

**Problem: How do we provide achieve the performance of containers with the security of VMs?**

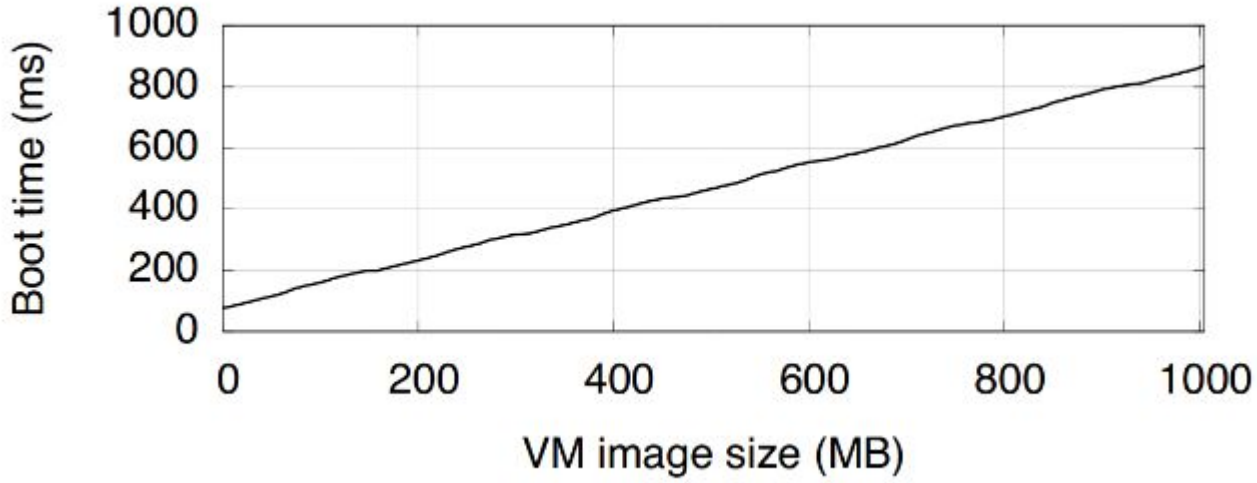
# Efficient VM requirement …

Provide efficient virtualization on top of hypervisors with the following aim:

- Lightweight
- Fast instantiation
- Capability of running hundreds of instances
- Pause / unpause

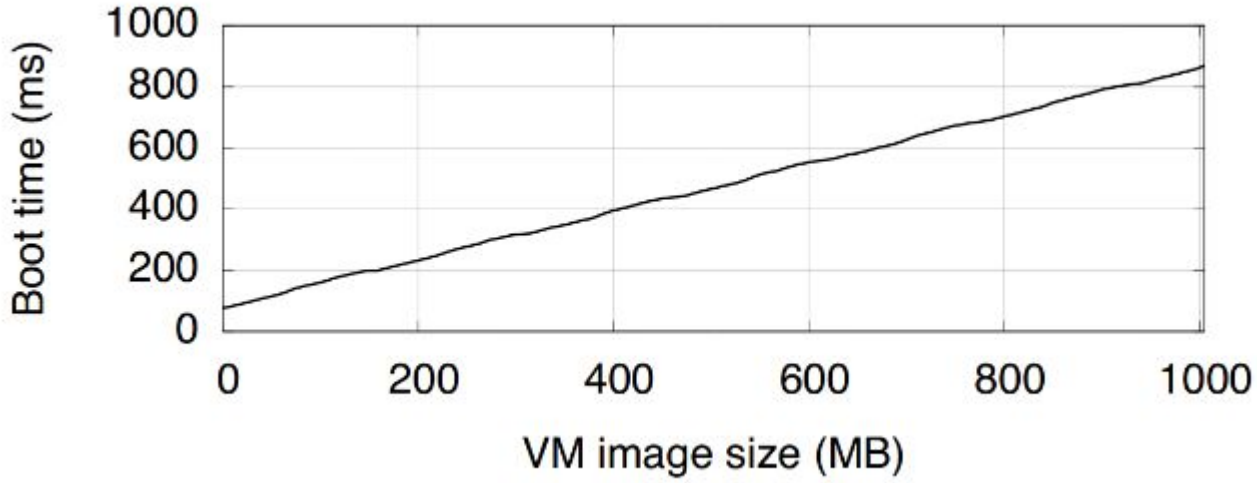
# Observation

- Size of guest VMs: limits both scalability and virtualization performance



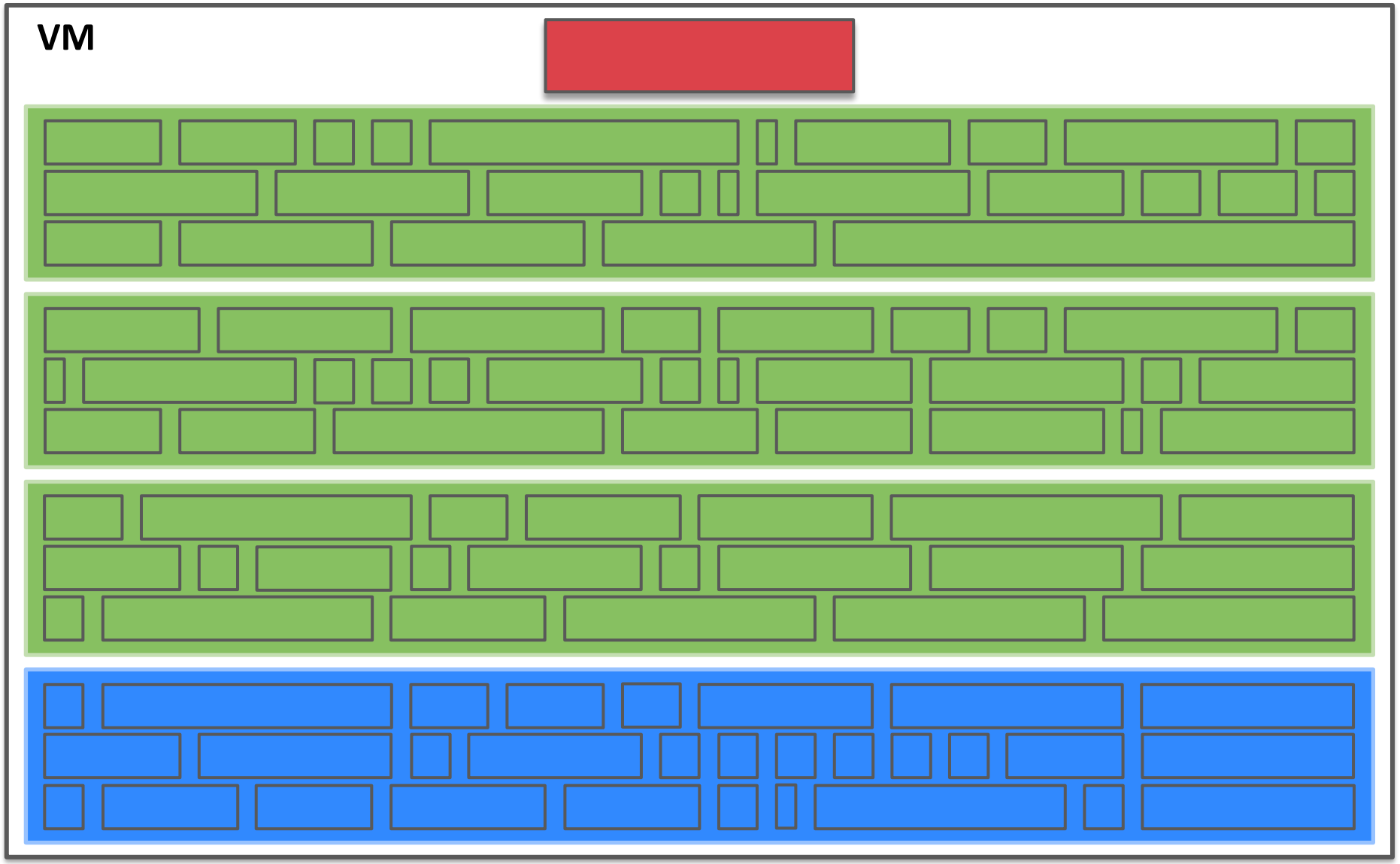
# Observation

- Size of guest VMs: limits both scalability and virtualization performance




- Most containers and VMs run one application at a time
  - Goal: **Minimize the size of the guest OS**

# Standard VM: application on top of distro




User Application

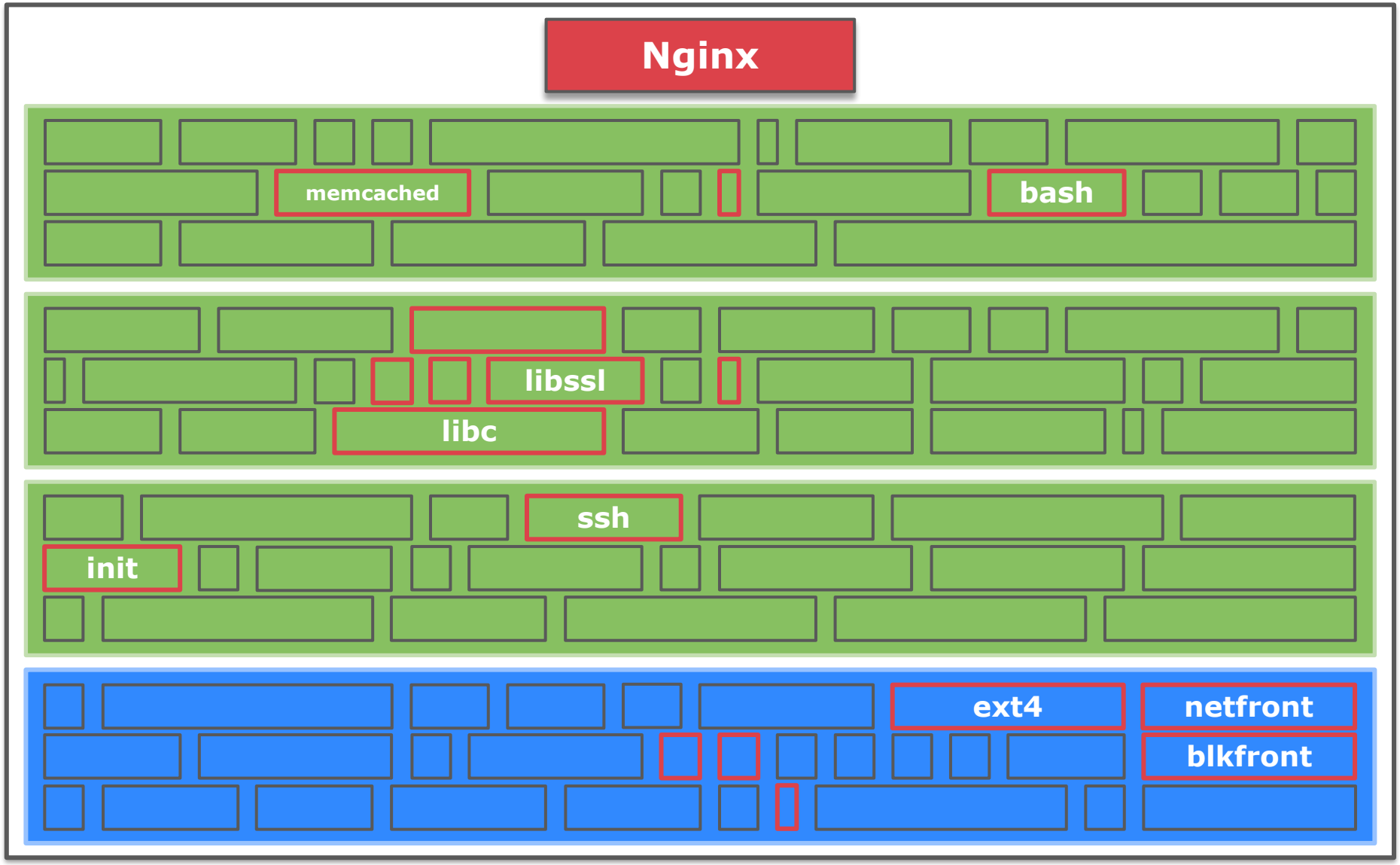
3rd Party Applications

Libraries

Services

Kernel

# Most of the VM is not used




User Application

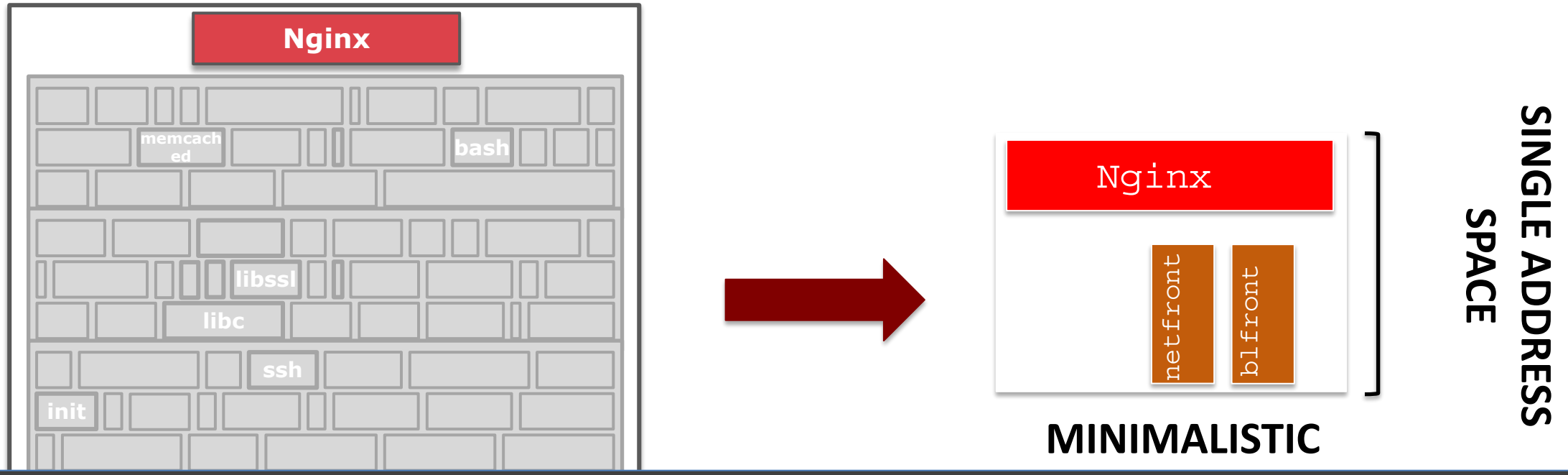
3rd Party Applications

Libraries

Services

Kernel

# Unikernels: single app + minimalistic OS



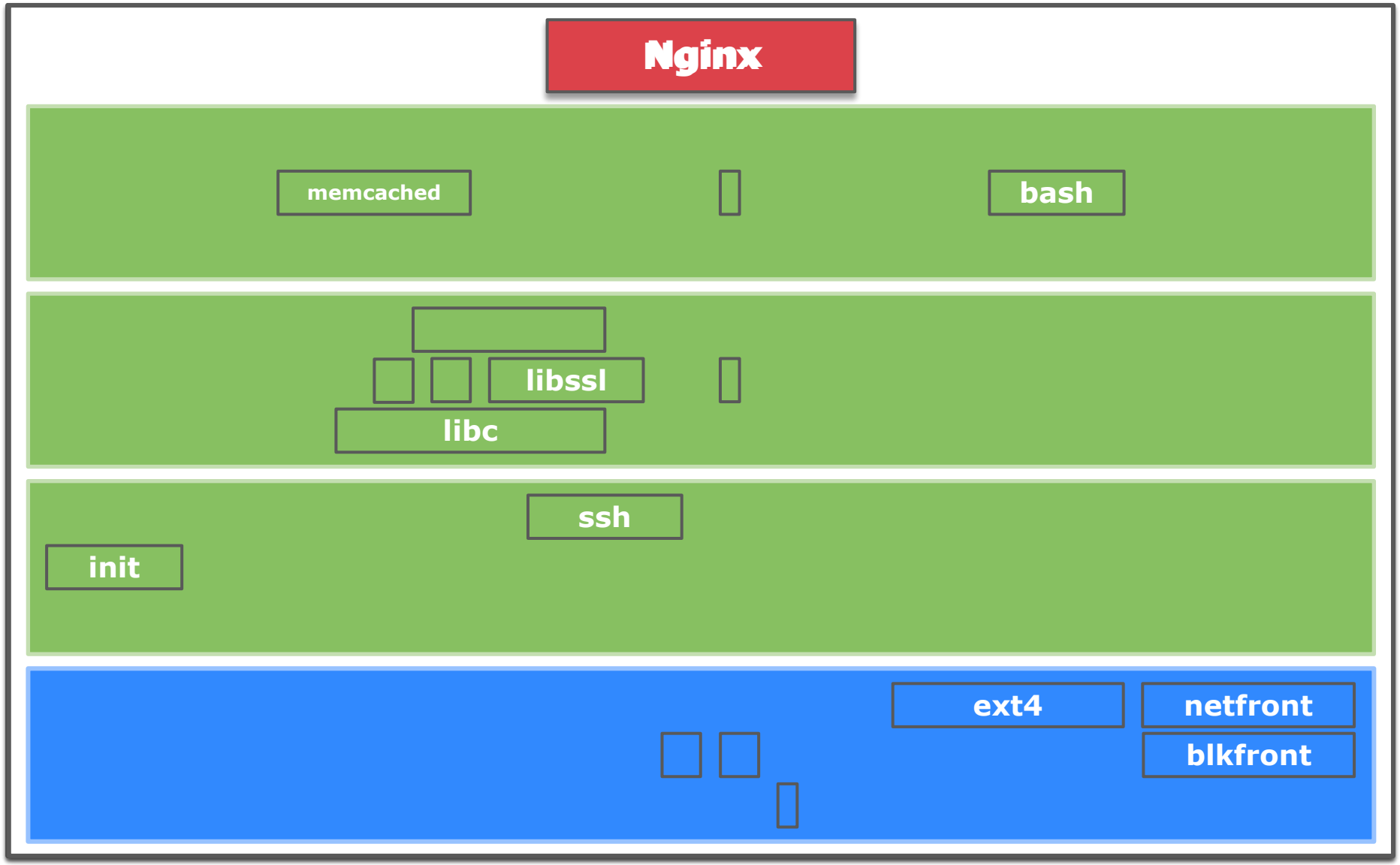

**Unikernels are lightweight:**

- **Daytime -** 480KB disk, 3.4MB RAM
- **Minipython -** 3.52MB disk, 8MB RAM
- **TLS termination proxy** – 3.58MB disk, 8MB RAM

# Unikernels

- Tiny VMs where a minimalistic OS is directly linked with specific application
  - Size of the VM is always small
- Example:
  - ClickOS: running custom Click modular router
  - Mirage: Creates a minimalistic app + OS combination for given OCaml app.
  - MiniOS: toy guest OS

# Tinyx: Small Linux distro for target app.

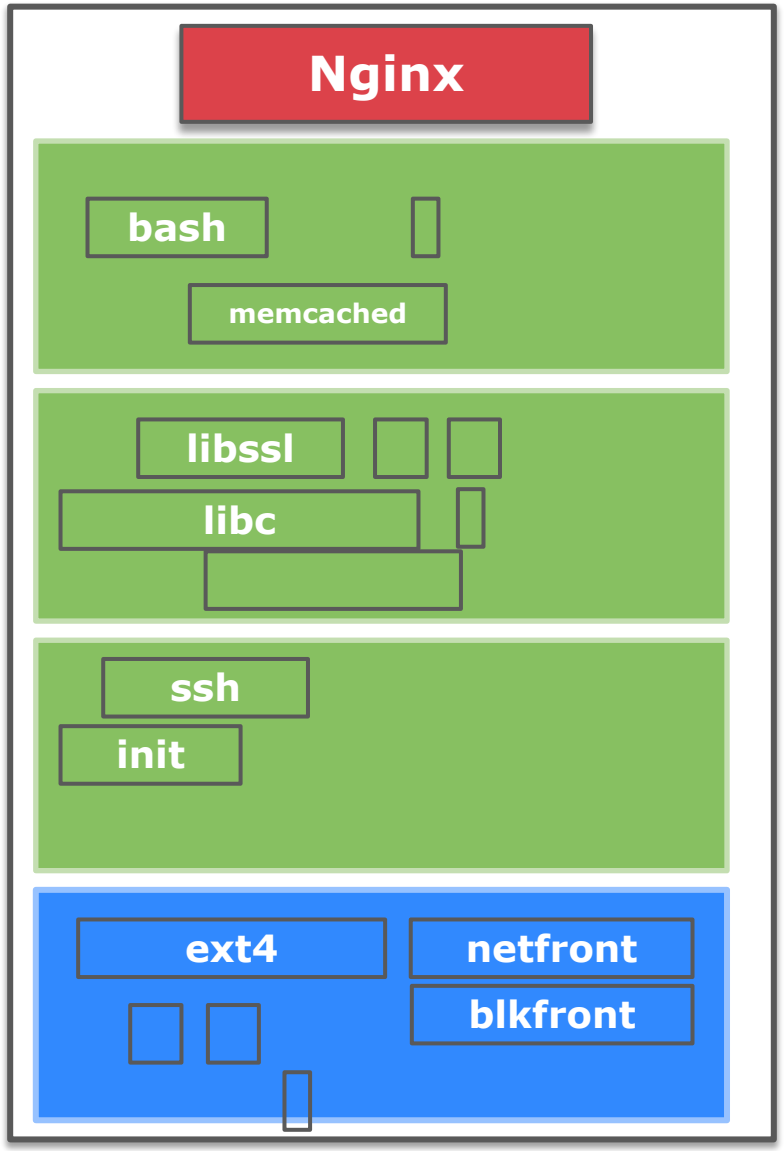



Find dependencies
- objdump
- dpkg

Install app & deps
- OverlayFS
- Cleanup

Small kernel:
- Remove drivers
- User config opts

# Tinyx: Small Linux distro for target app.




Tinyx VMs are also lightweight:

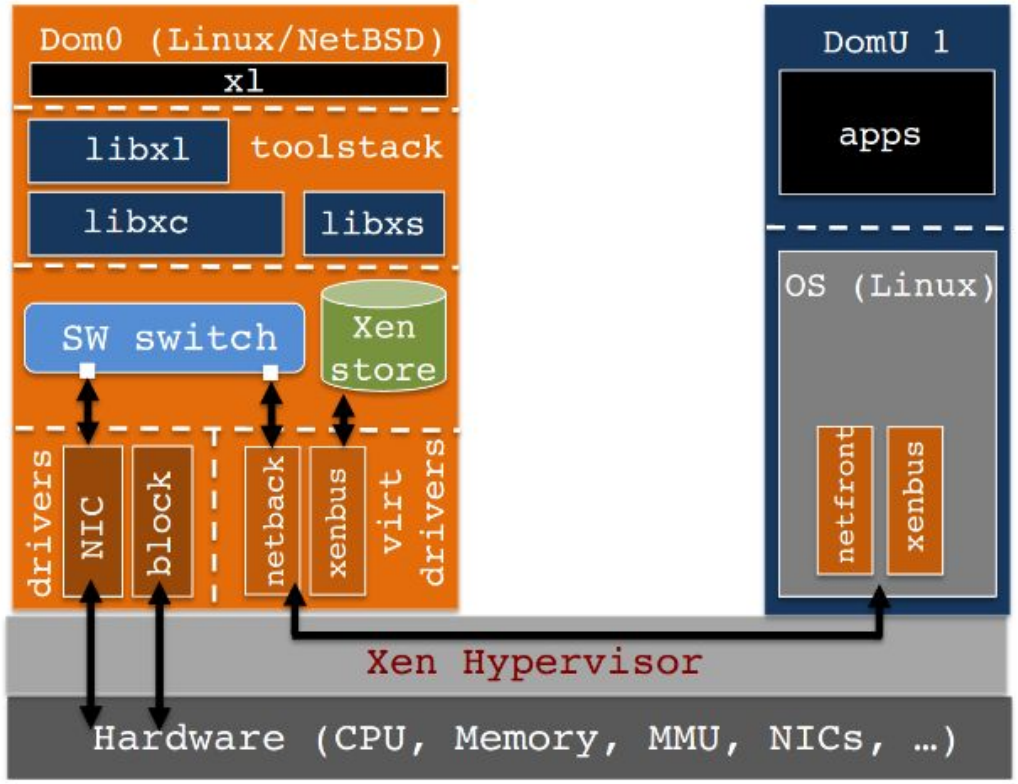
- Kernel: 1.5MB (compared to 8MB)
- Image size – 10-30MB (compared to 1GB).
- Boot: 200ms instead of 2s.

# Question …

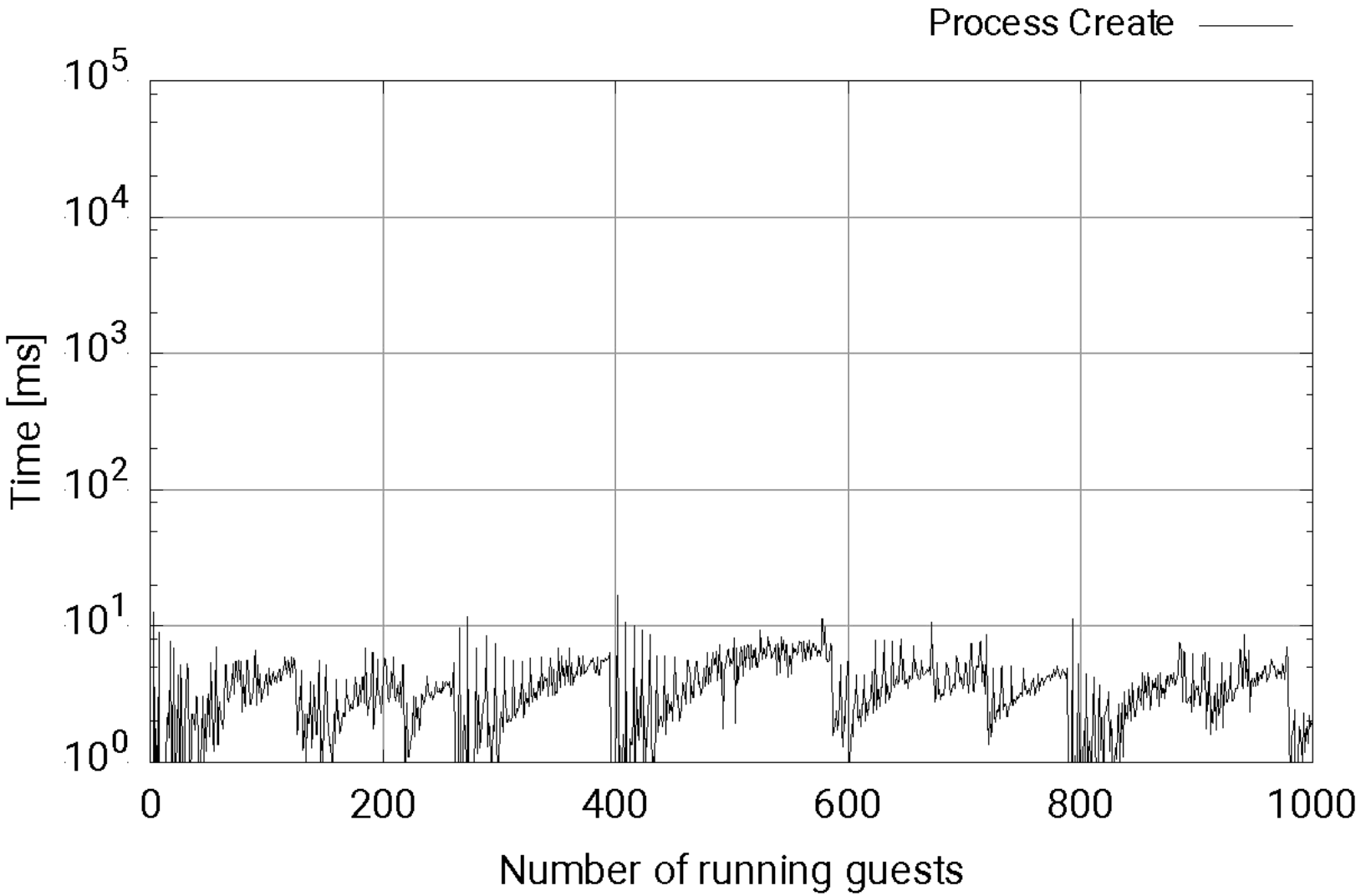
**Are lightweight VMs enough for good performance?**

# Look into Xen …

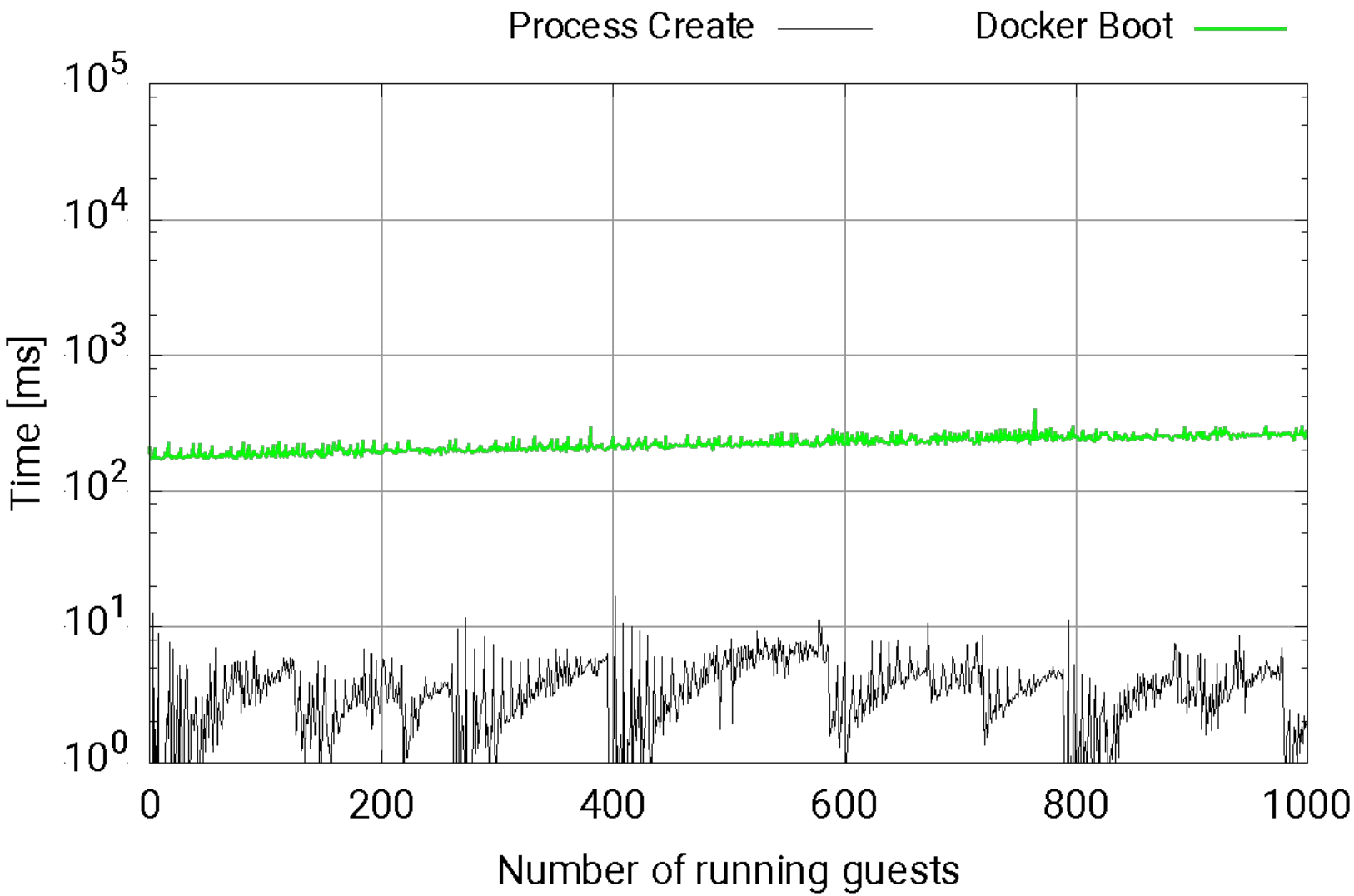
- Type 1 hypervisor
- Manages basic resources (CPU, memory, etc.)
- Launches most privileged driver domain: dom0
  - Direct access to hardware
  - Manage the hypervisor and launches unprivileged guest domains called DomUs



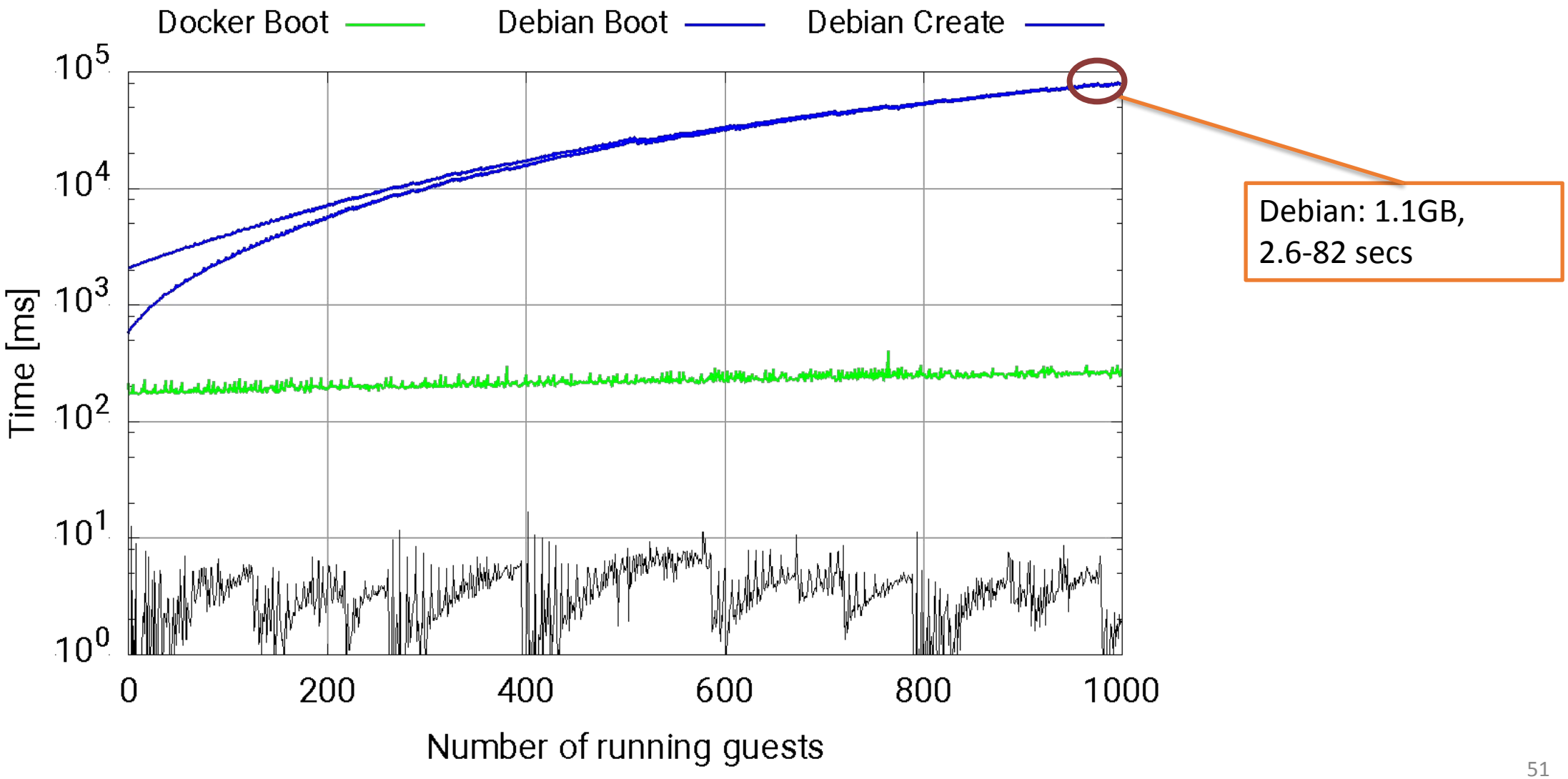
# VM, container, and process creation times



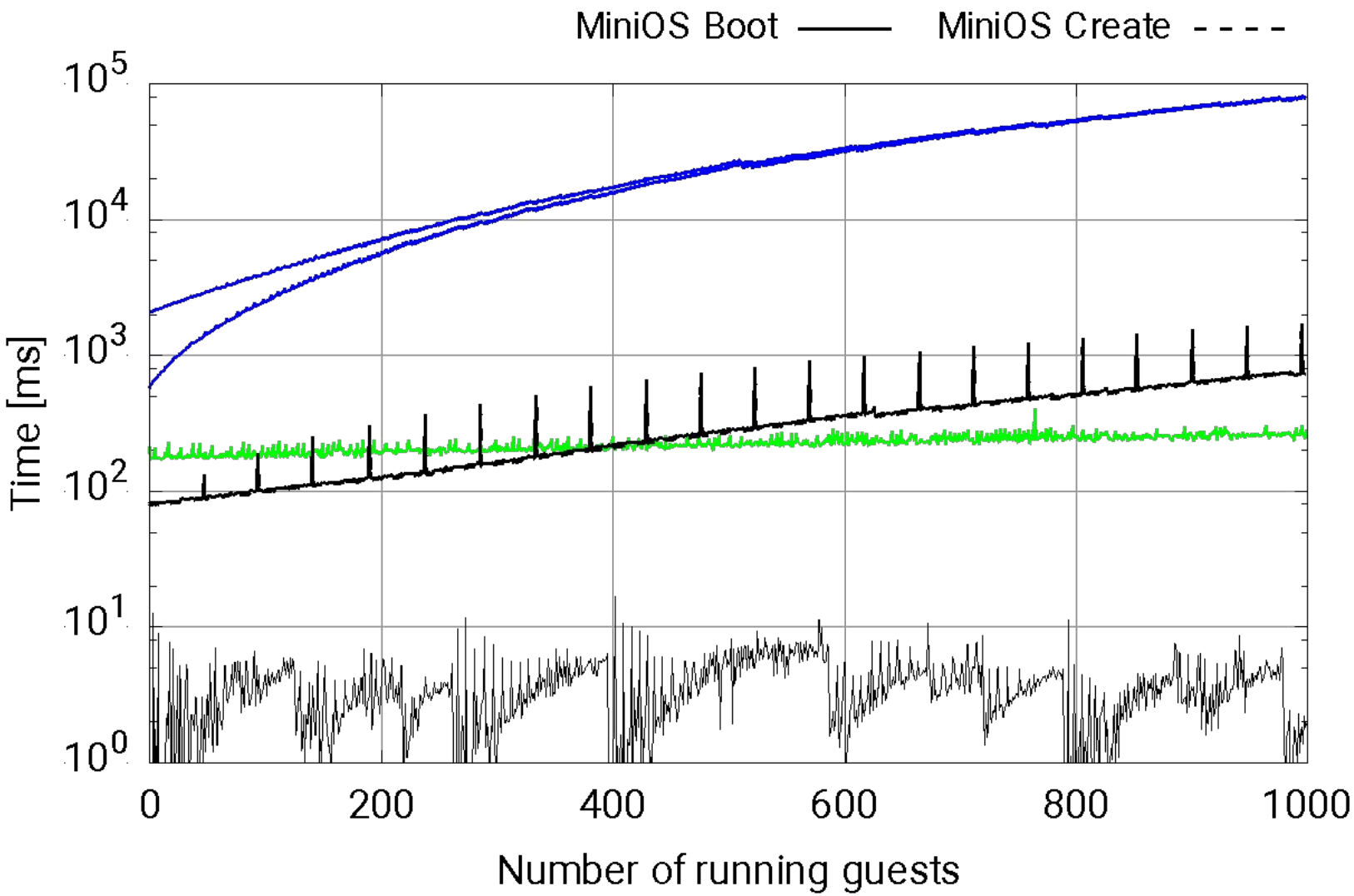
# VM, container, and process creation times



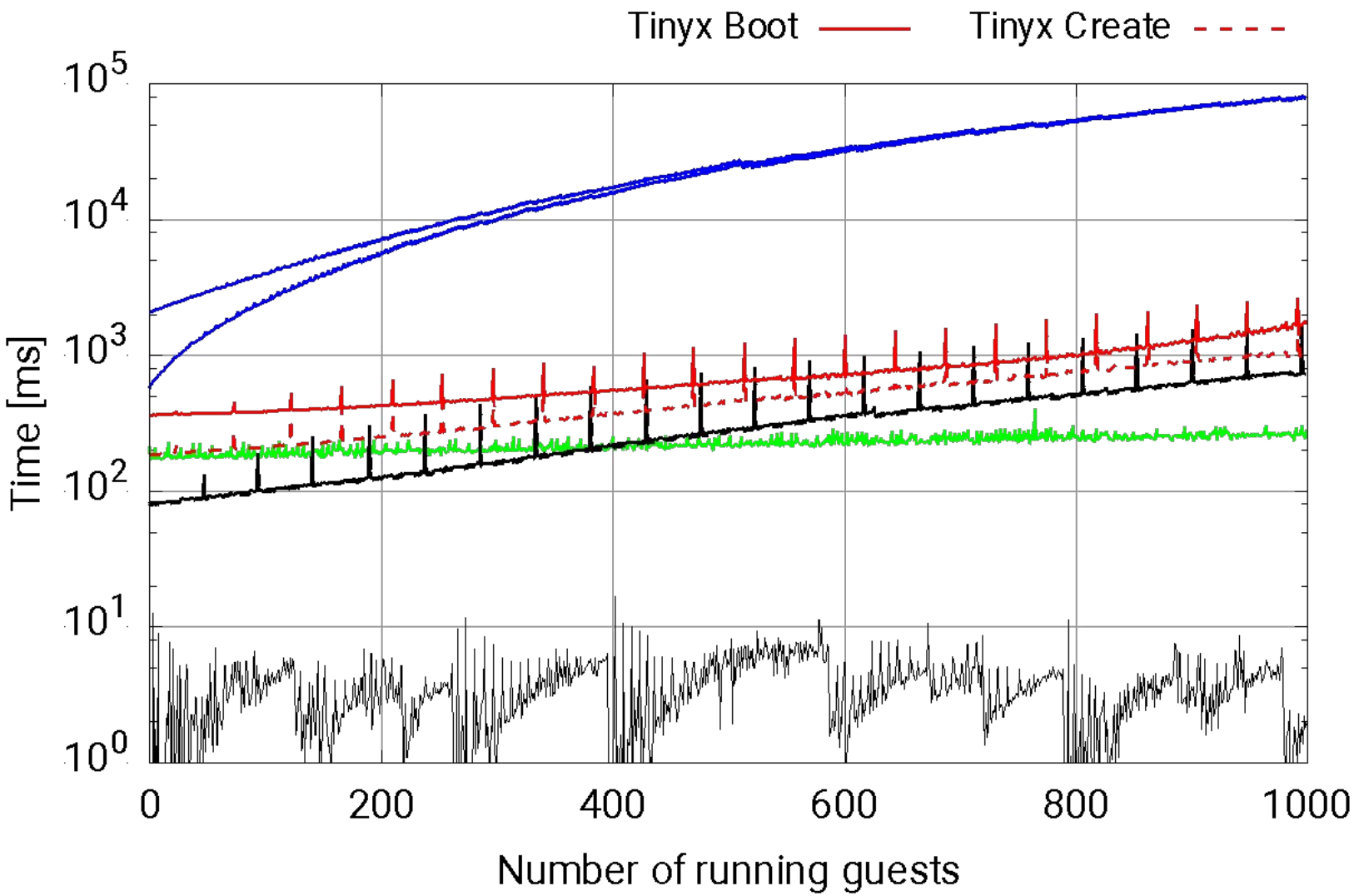
# VM, container, and process creation times



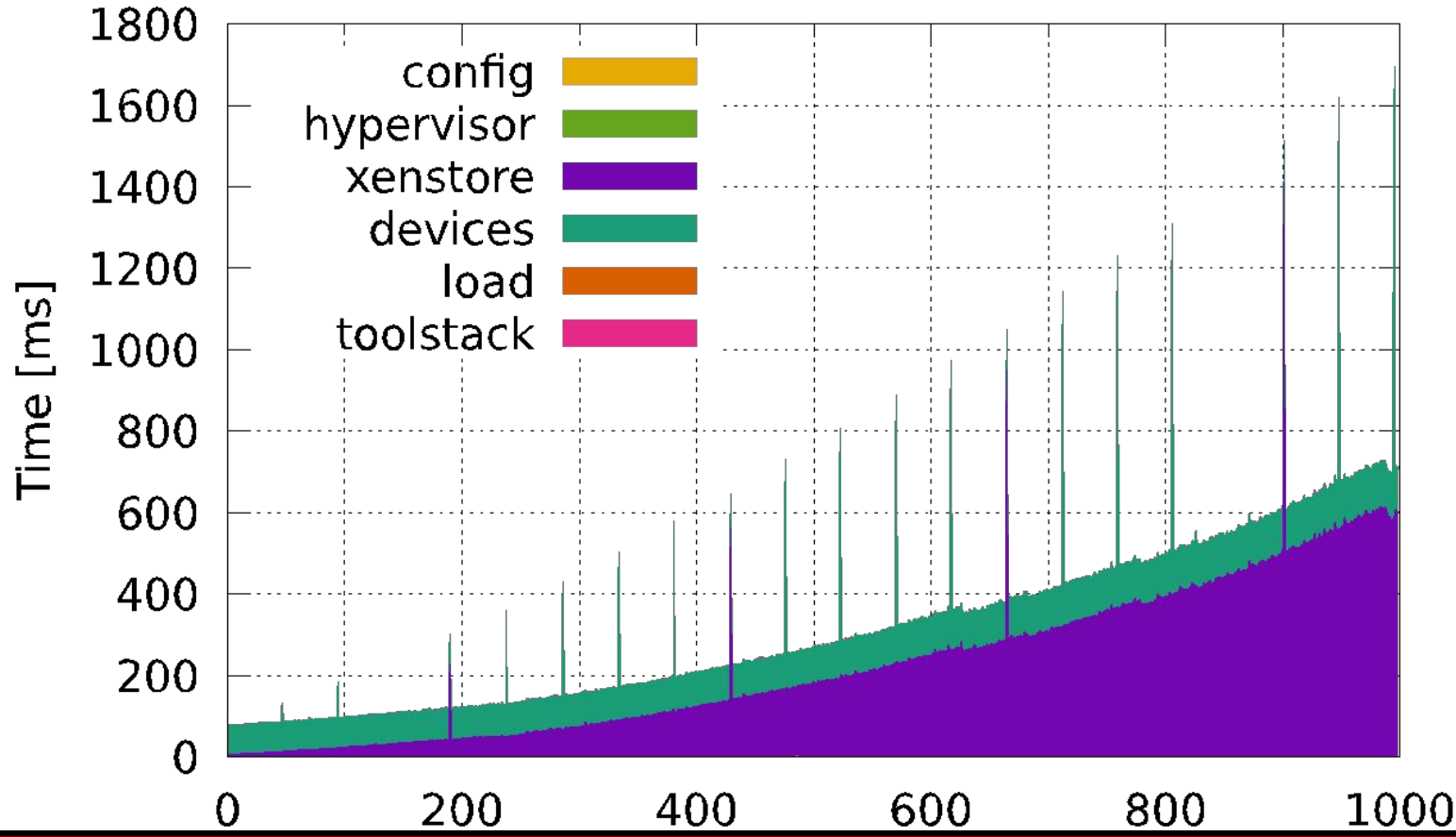
# VM, container, and process creation times



# VM, container, and process creation times



# Xen: Creation time breakdown (unikernel)

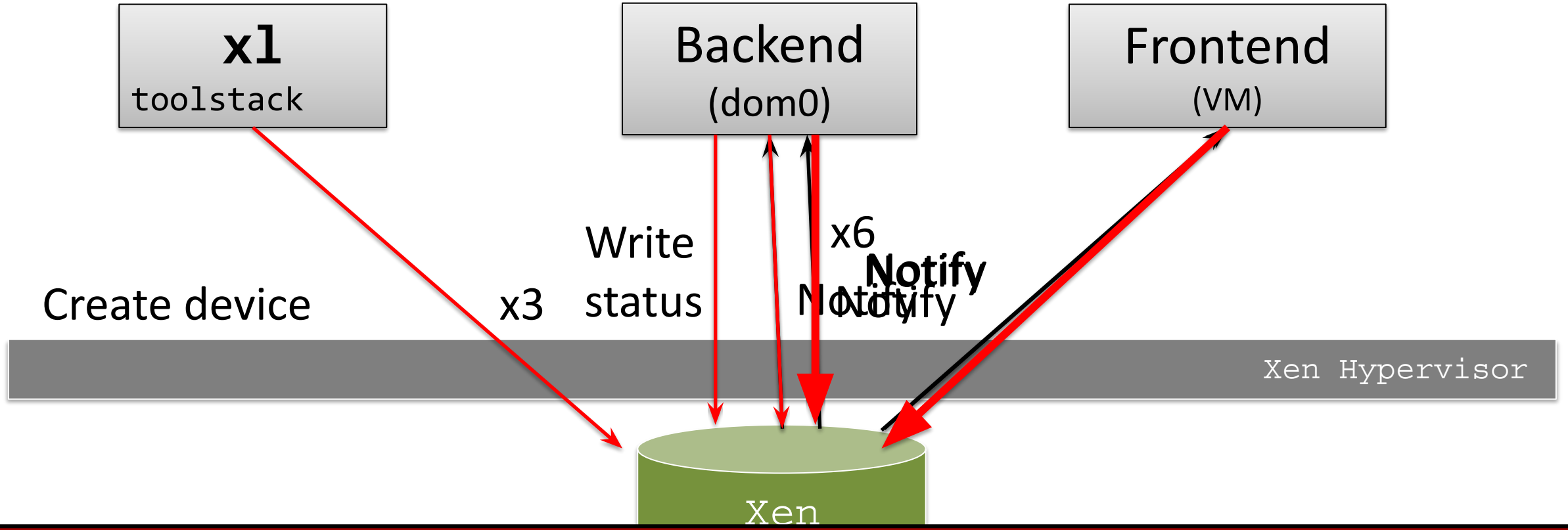



**Biggest culprits: XenStore and device creation**

# Xenstore overhead

- Message based protocol triggers multiple software interrupts and domain changes between Dom0 kernel and userspace, hypervisor and the guest
- Writing certain information (e.g., unique guest names) causes linear overhead
- Writing another information type requires writing data in multiple XenStore log records: uses transaction to ensure atomicity
- Xenstore logs every access to log files

# What is wrong with Xenstore?

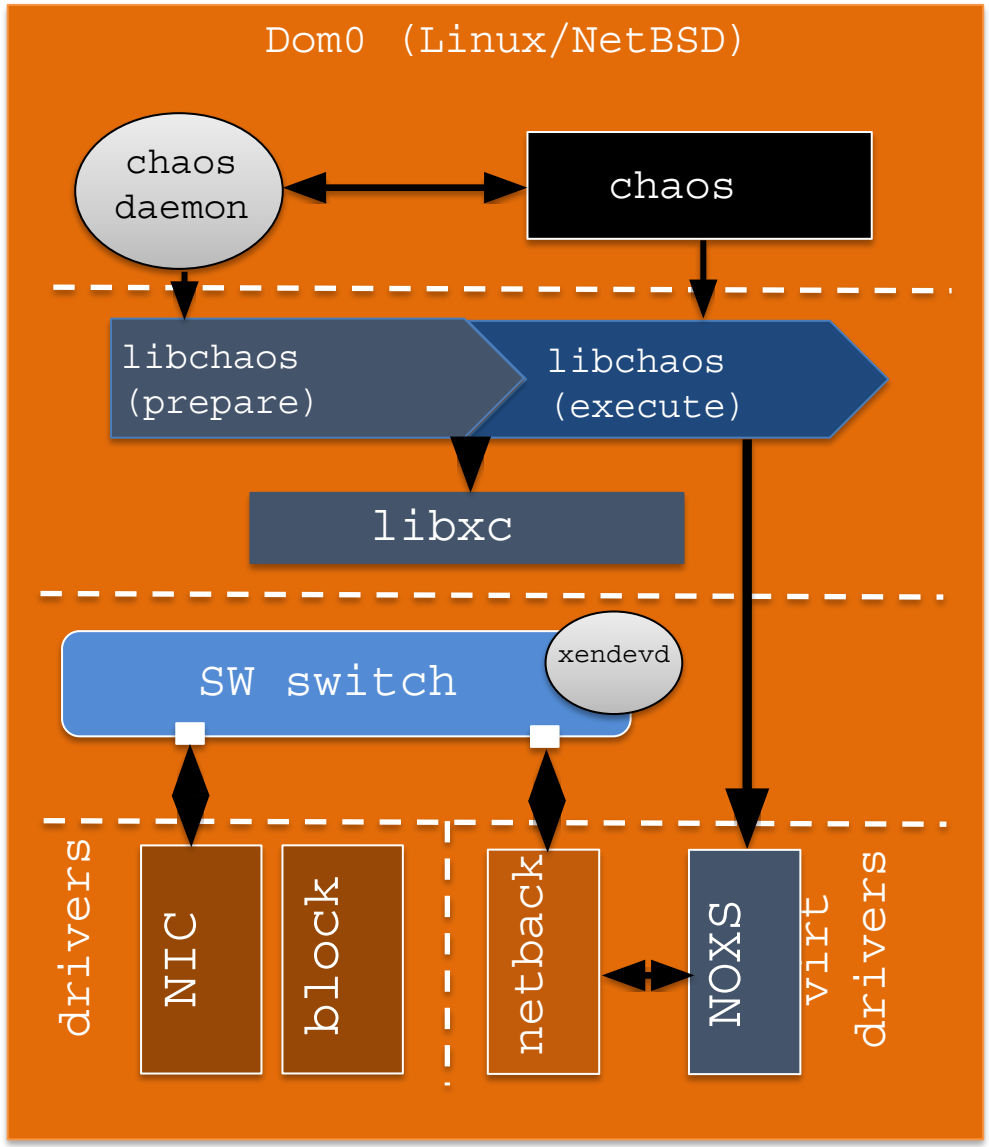



**More than 30 Xenstore entries are used per device, resulting in hundreds of XenStore accesses**

# LightVM: the alternative!

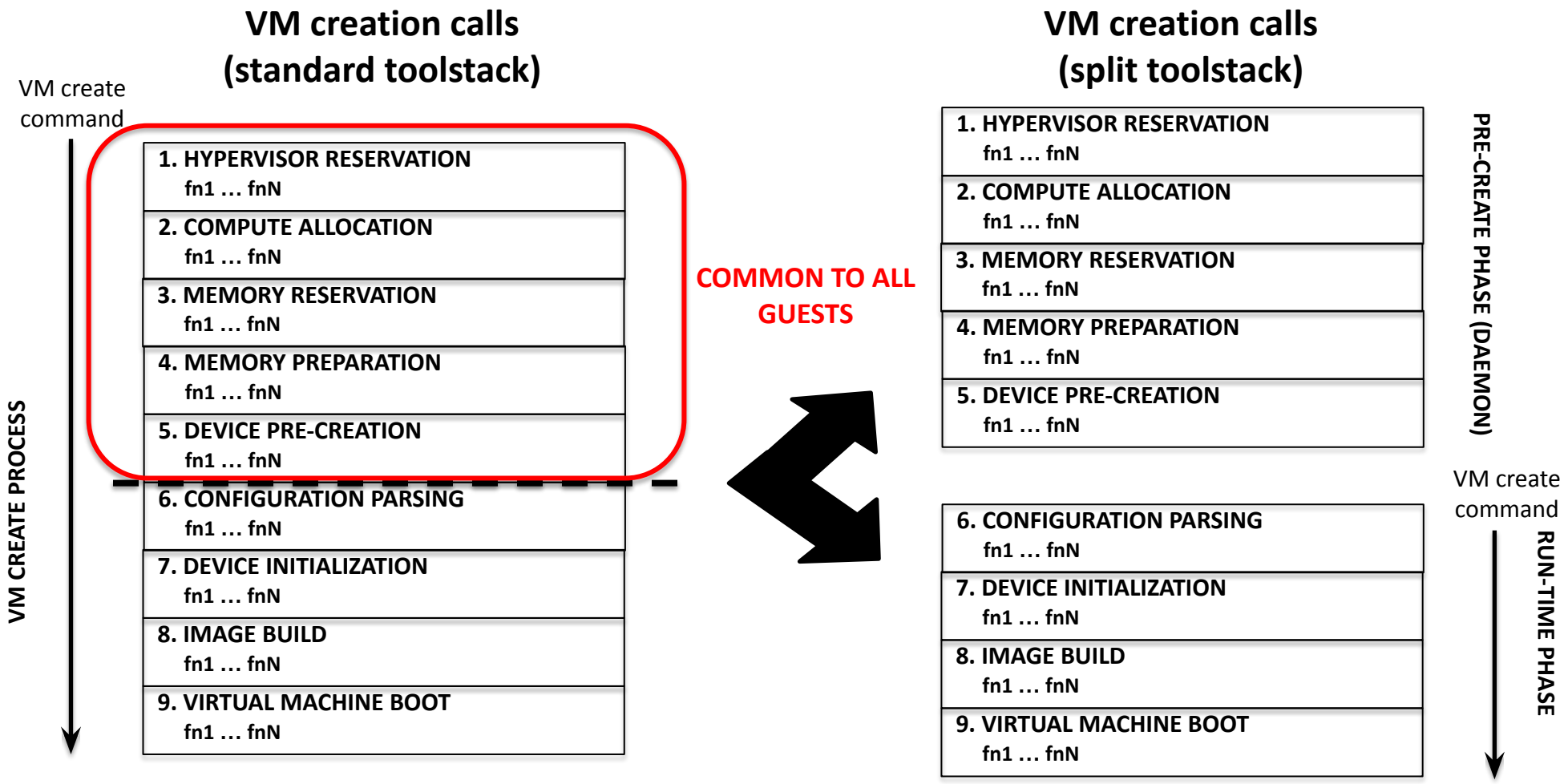
- Re-design of the basic Xen control plane
- No Xenstore for VM creation or boot, instead uses a lean driver (noxs)
- New virtualization toolstack as well (libchaos)

# LightVM architecture




1. **Chaos** – toolstack optimized for paravirtualized guests
2. **Split functionality**
3. **Noxs** - no XenStore

# Split toolstack



# Removes Xenstore

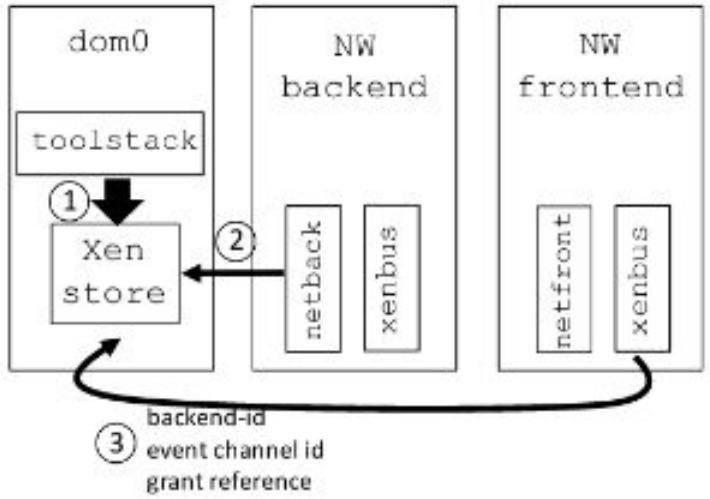
Xenstore is used to:

- Store data
- Communicate between guests
- Synchronization

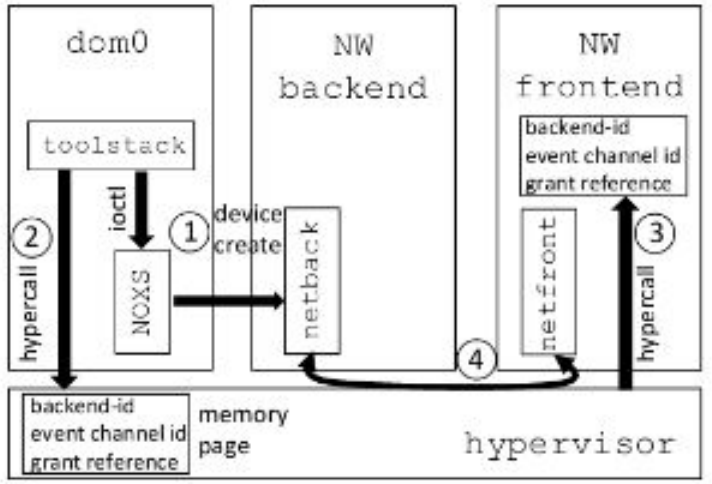
NoXS

**Shared memory**

**Event channels**

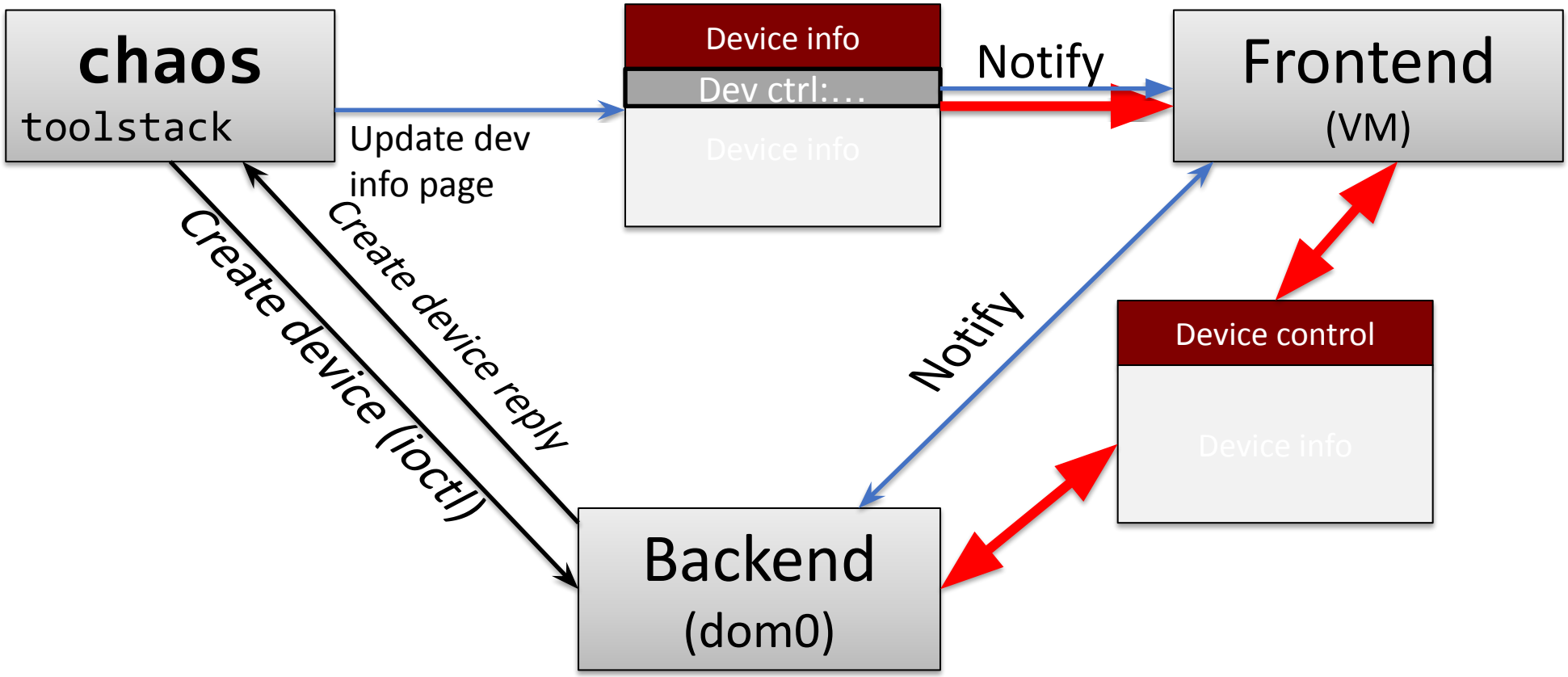



(a) XenStore

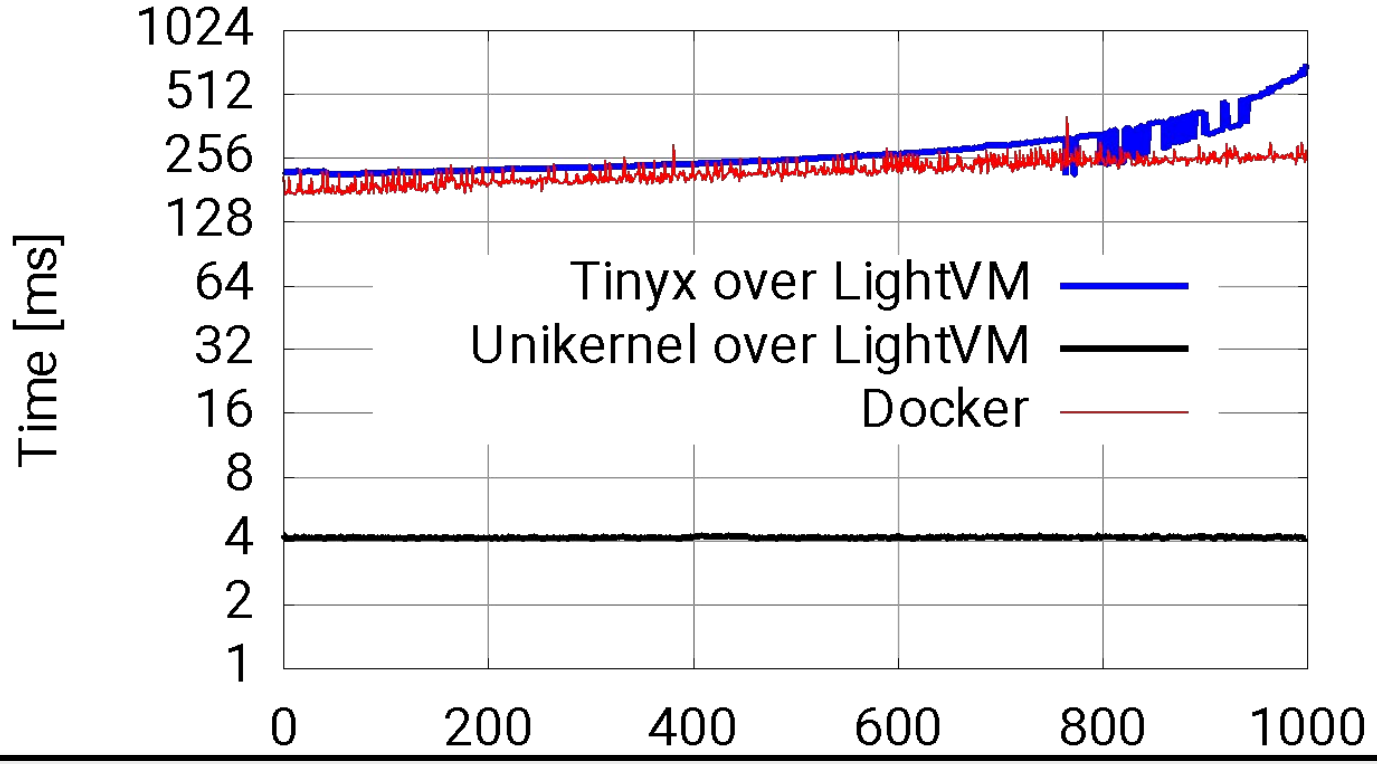



(b) noxs

# Noxs functioning



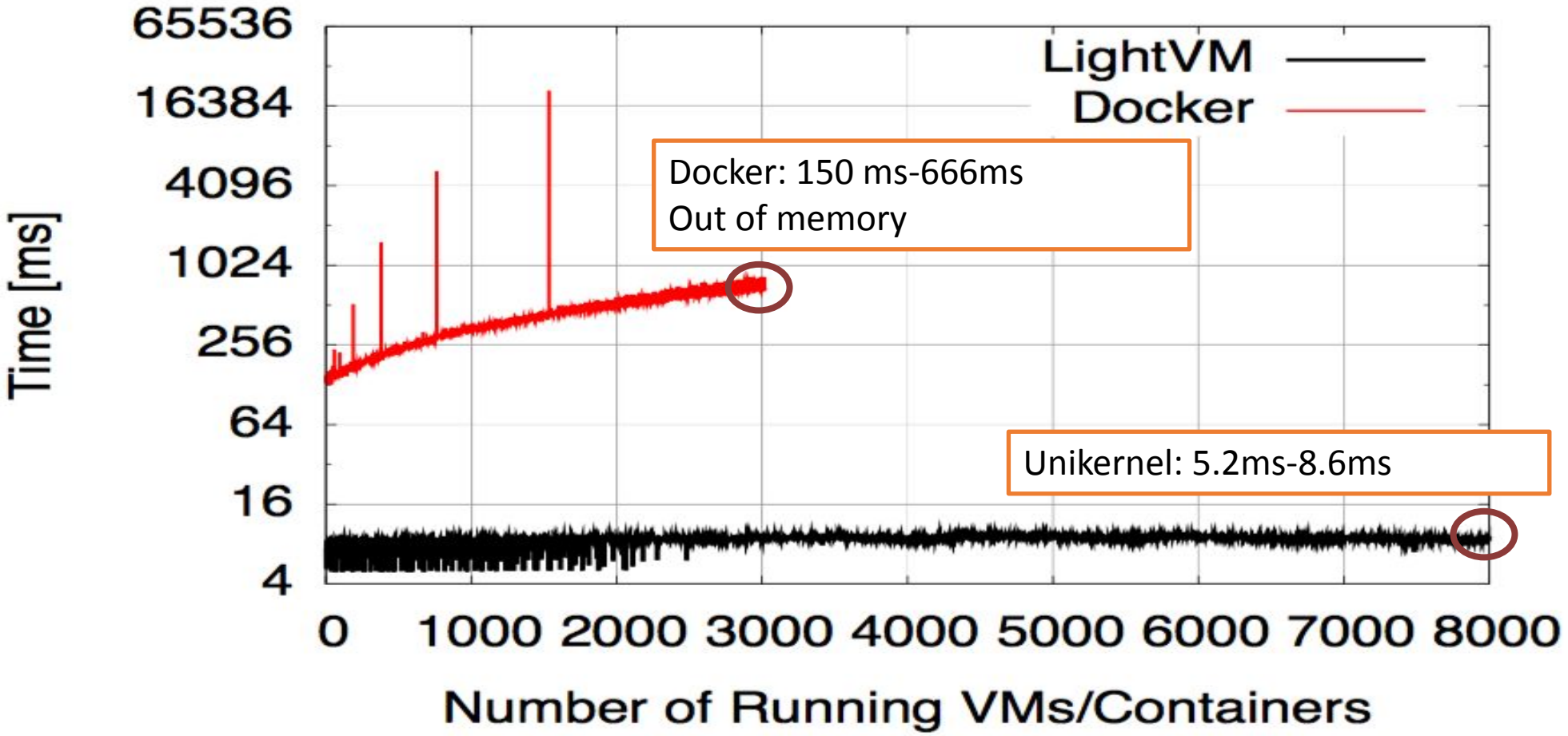
# Instantiation: Unikernel vs Tinyx




**Tinyx (minimalistic Linux) close to Docker times!**

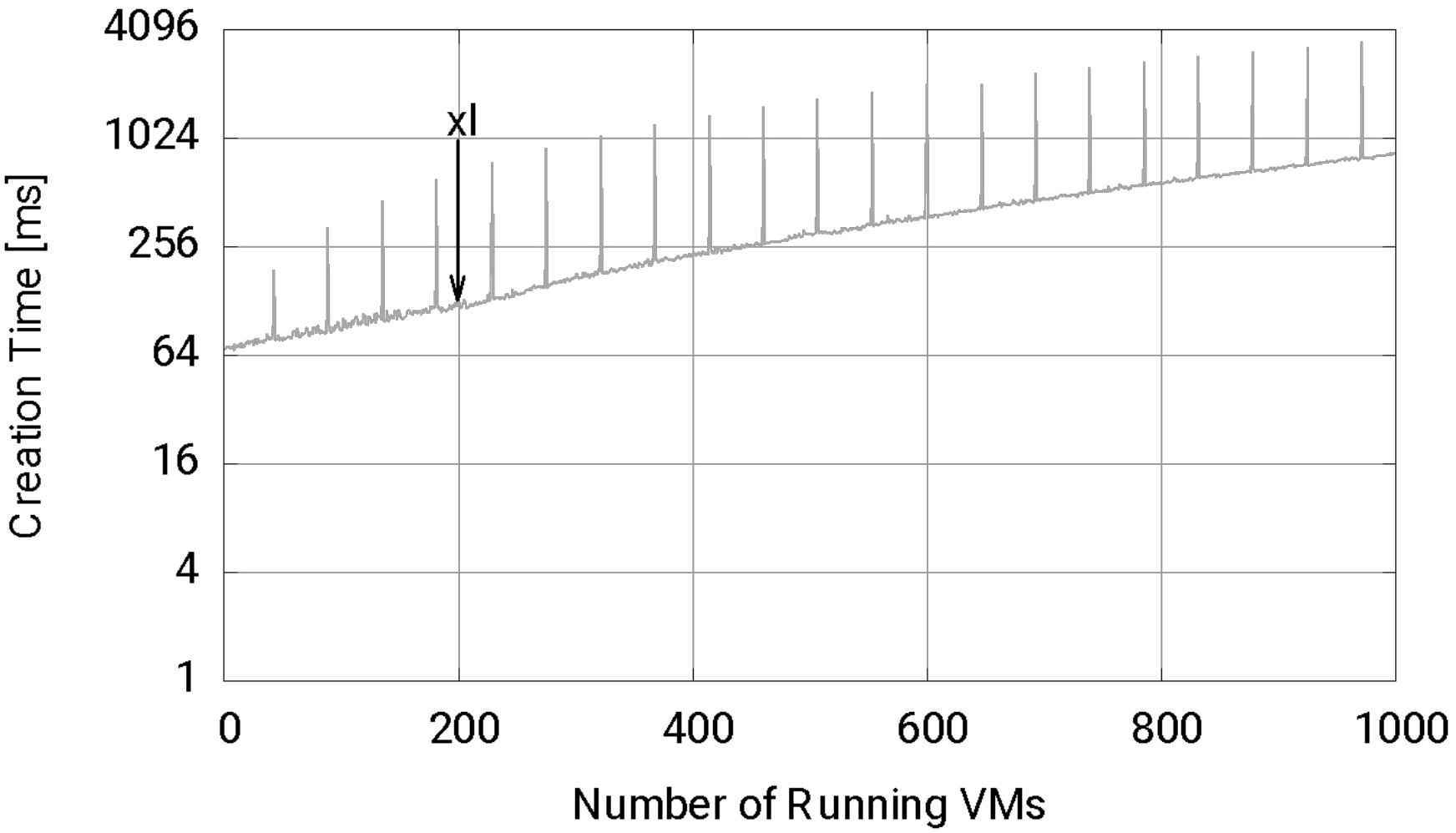
*Server: Intel Xeon E5-1630 v3 CPU@3.7GHz (4 cores), 128GB DDR4 RAM, Xen/Linux versions 4.8*

# Instantiation: high density (noop unikernel)



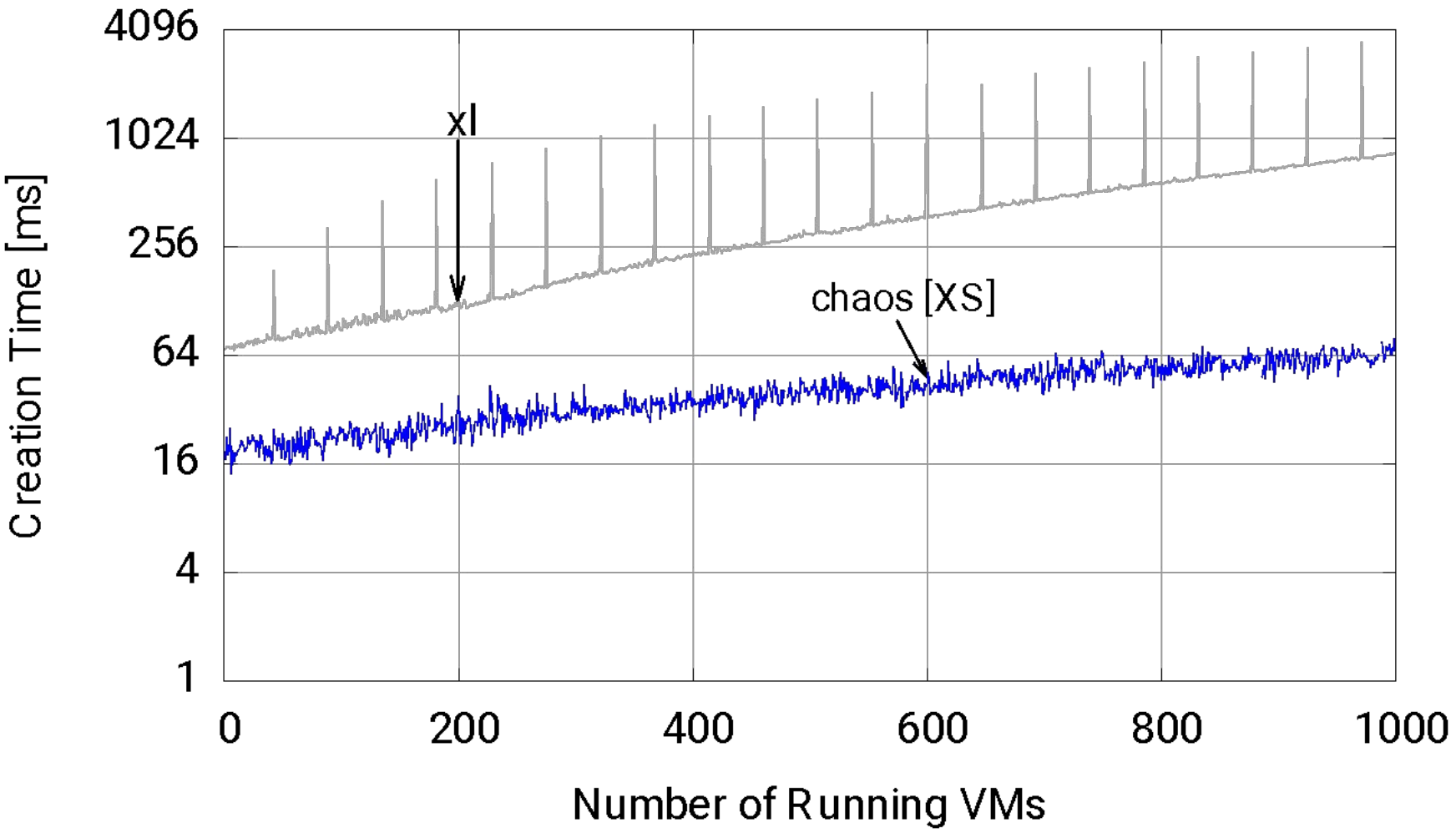

*Server: 4 x AMD Opteron 6376 CPU@2.3GHz (64 cores total), 128GB DDR3 RAM, Xen/Linux versions 4.8*

# Understanding LightVM components



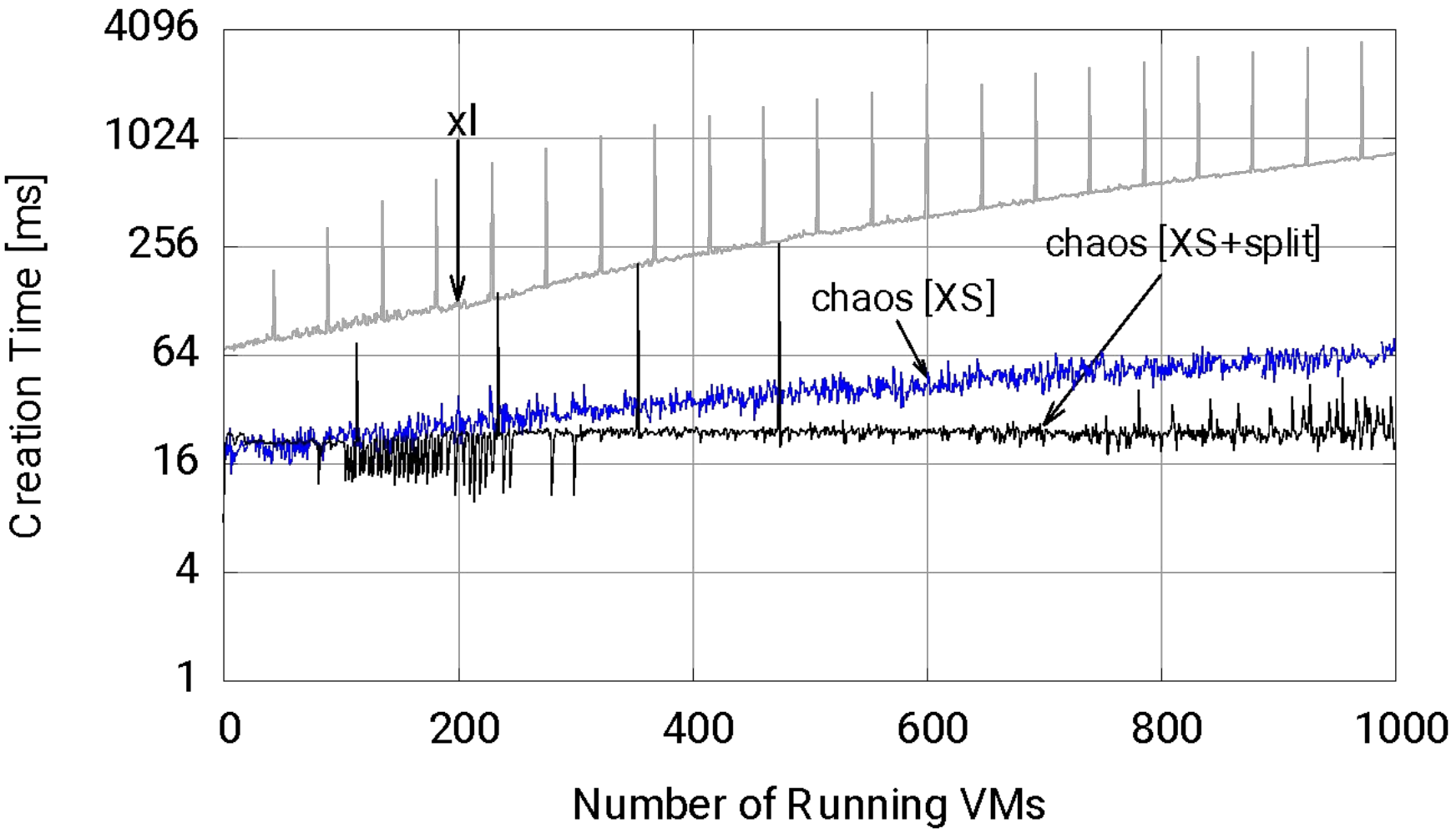

*Server: Intel Xeon E5-1630 v3 CPU@3.7GHz (4 cores), 128GB DDR4 RAM, Xen/Linux versions 4.8*

# Understanding LightVM components



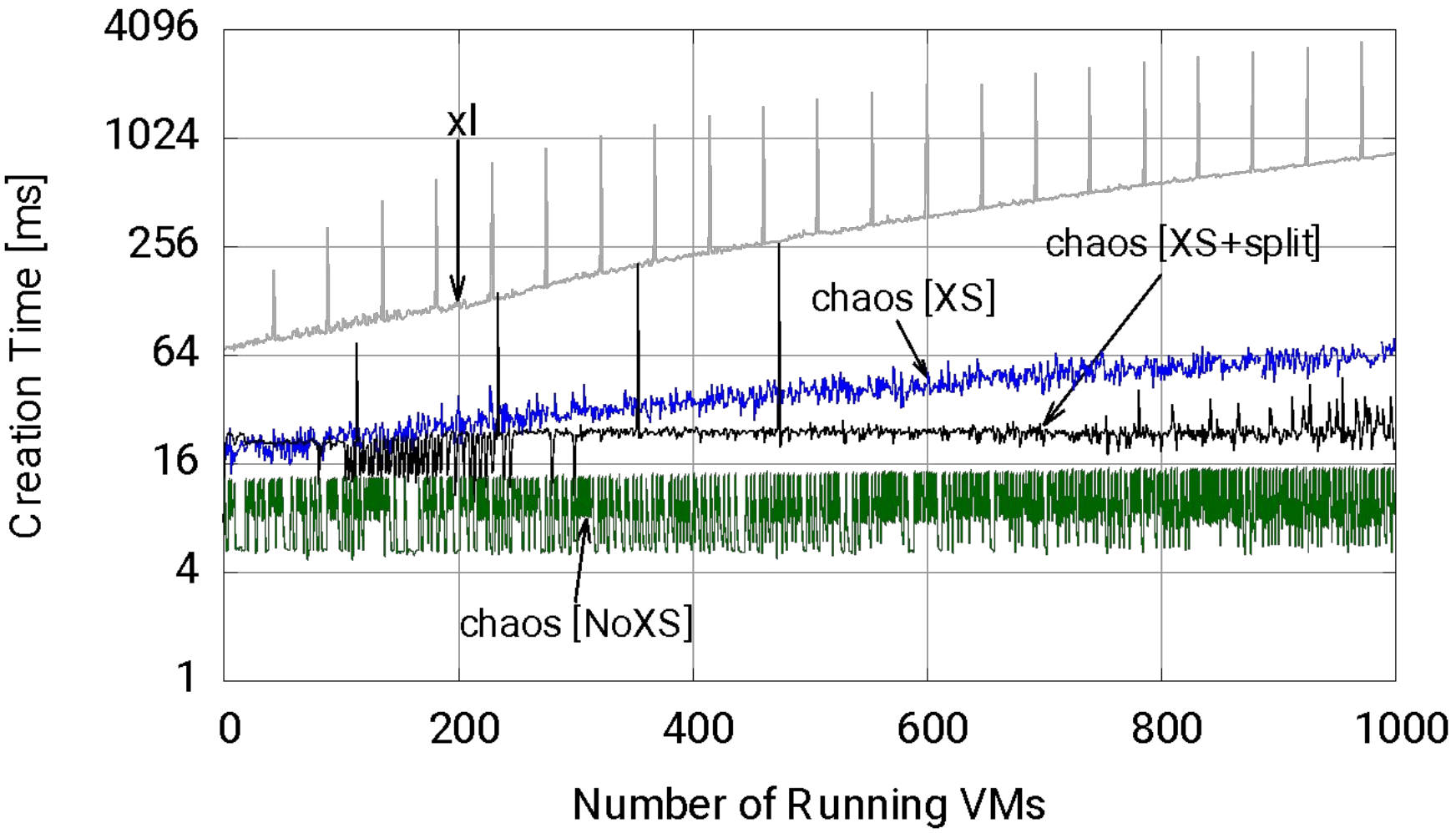

*Server: Intel Xeon E5-1630 v3 CPU@3.7GHz (4 cores), 128GB DDR4 RAM, Xen/Linux versions 4.8*

# Understanding LightVM components



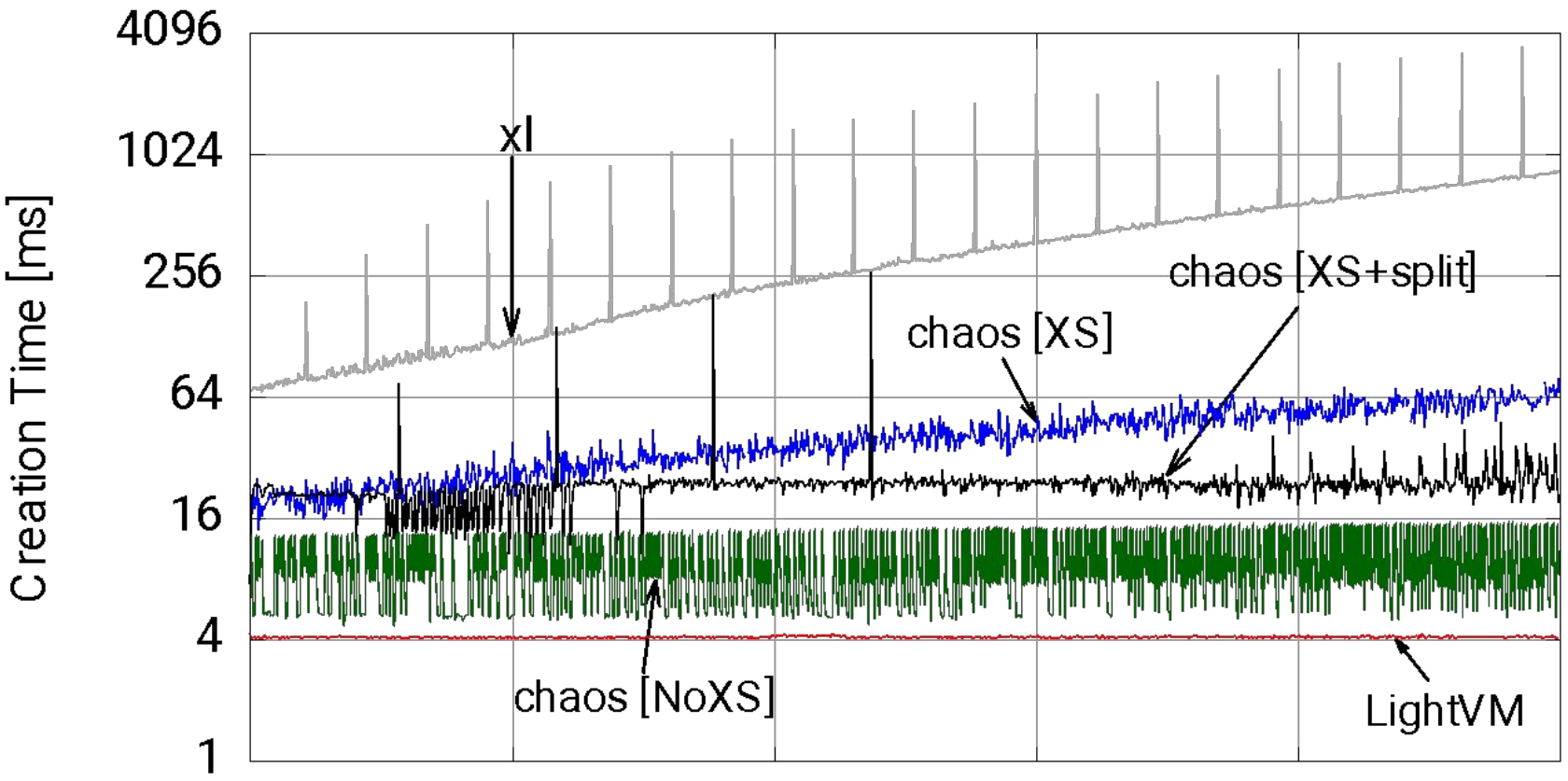

*Server: Intel Xeon E5-1630 v3 CPU@3.7GHz (4 cores), 128GB DDR4 RAM, Xen/Linux versions 4.8*

# Understanding LightVM components




*Server: Intel Xeon E5-1630 v3 CPU@3.7GHz (4 cores), 128GB DDR4 RAM, Xen/Linux versions 4.8*

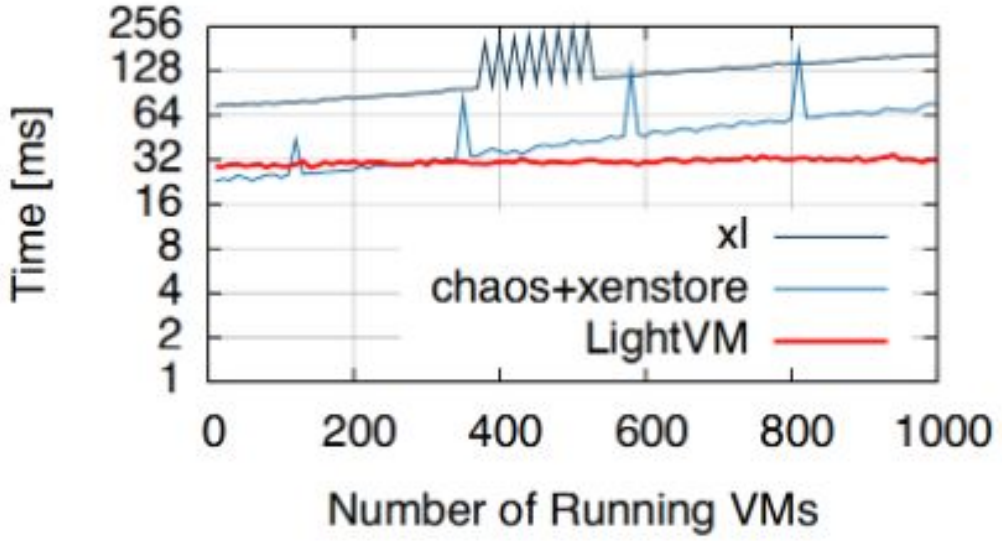
# Understanding LightVM components



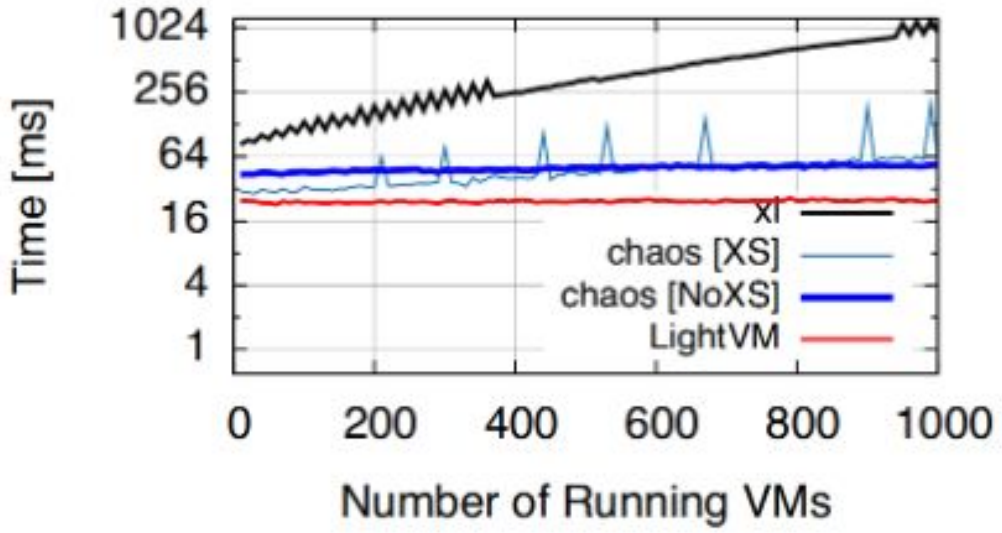

**Note: 2.3 ms with all optimizations and no devices**

*Server: Intel Xeon E5-1630 v3 CPU@3.7GHz (4 cores), 128GB DDR4 RAM, Xen/Linux versions 4.8*

# Checkpointing



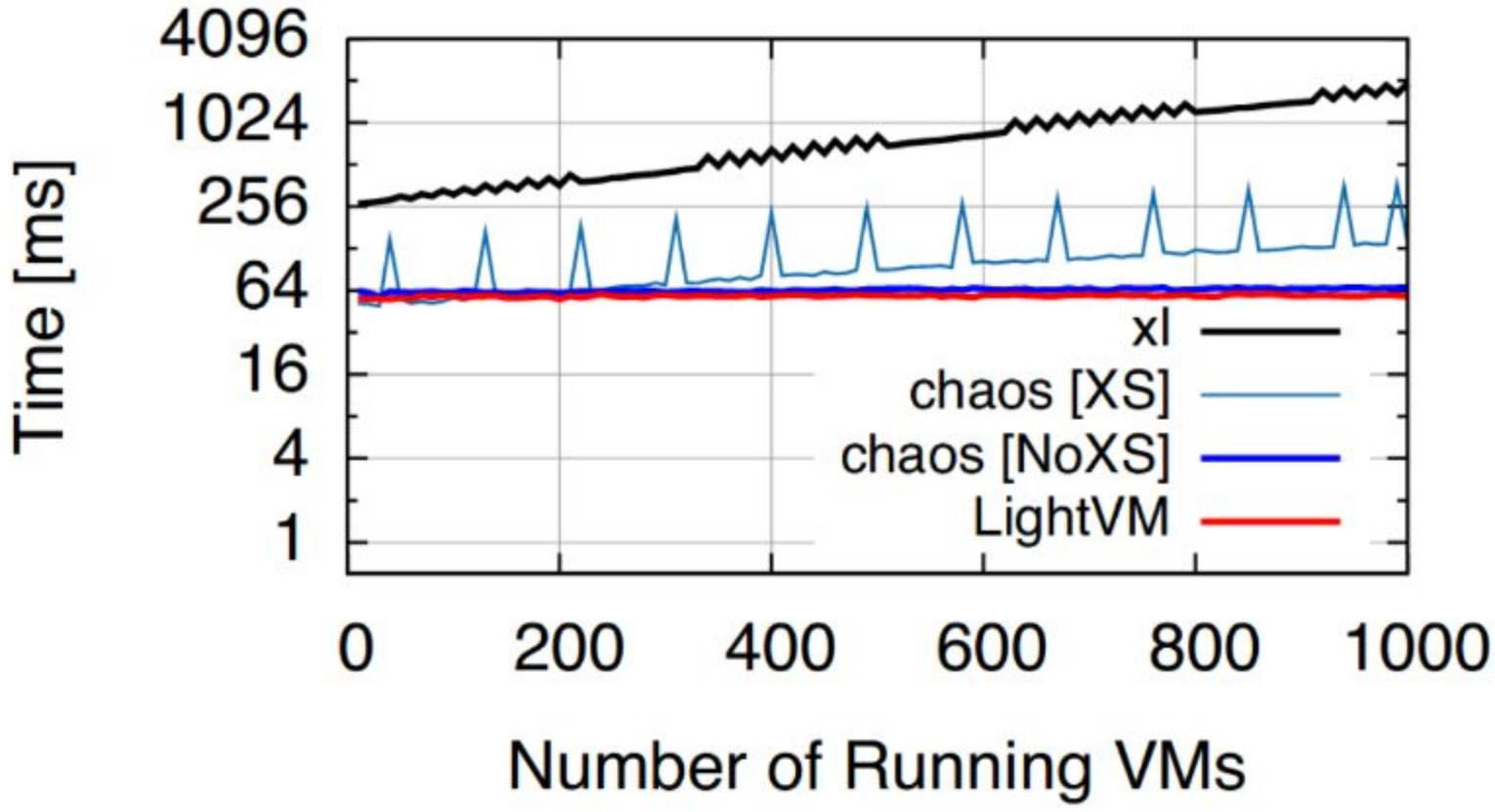

(a) Save




(b) Restore

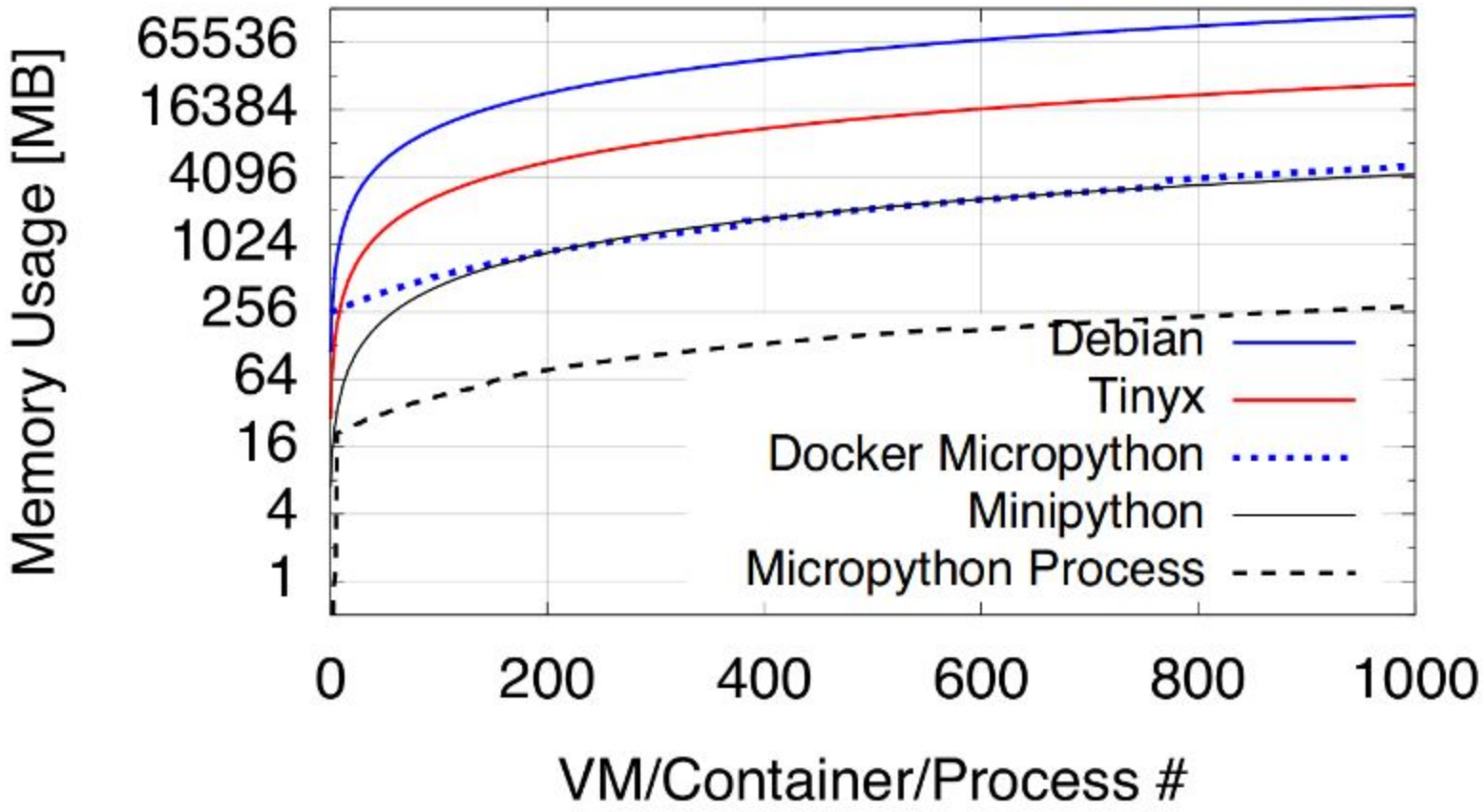
*Server: Intel Xeon E5-1630 v3 CPU@3.7GHz (4 cores), 128GB DDR4 RAM, Xen/Linux versions 4.8*

# Migration



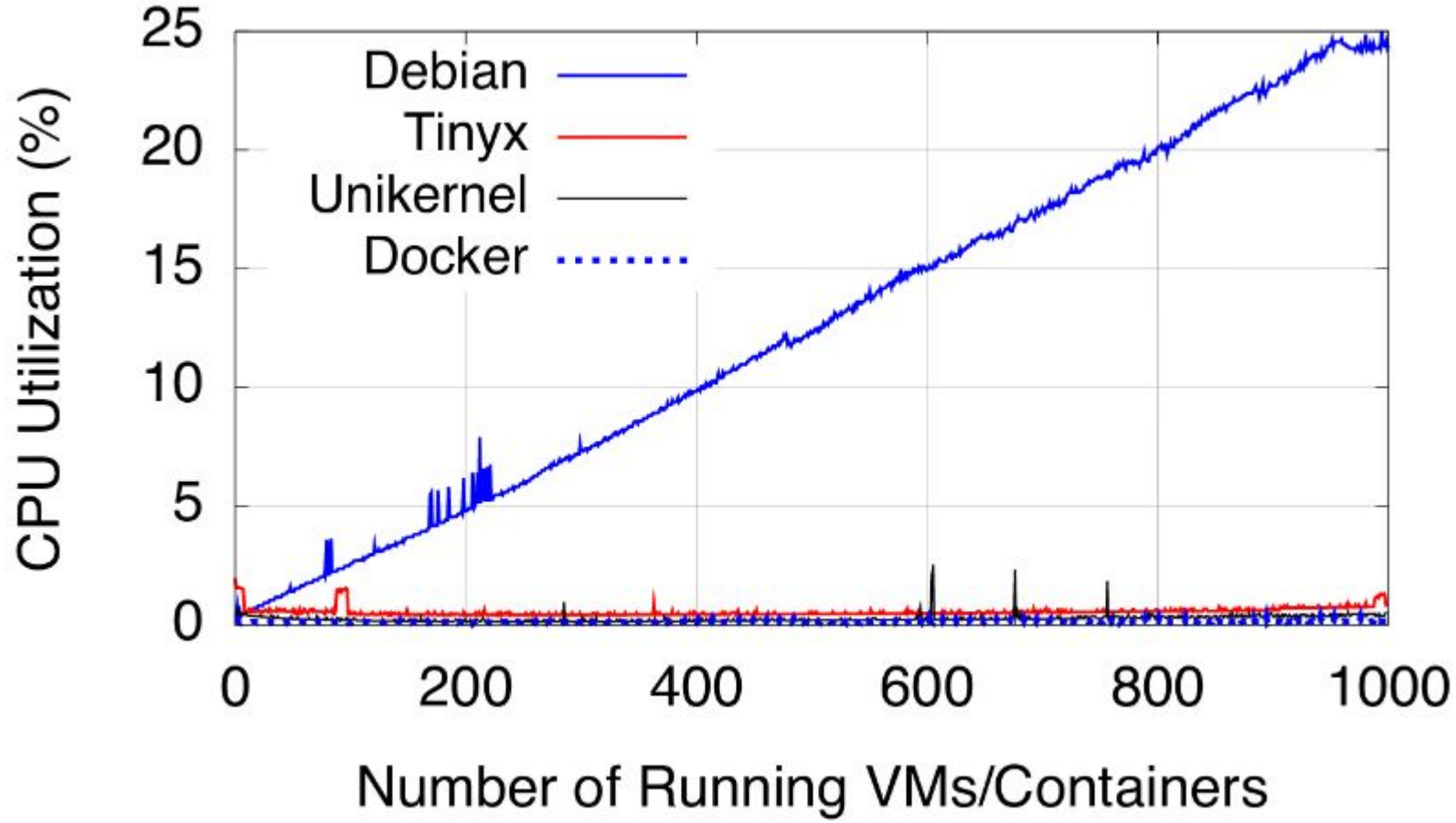

*Server: Intel Xeon E5-1630 v3 CPU@3.7GHz (4 cores), 128GB DDR4 RAM, Xen/Linux versions 4.8*

# Memory footprint




*Server: Intel Xeon E5-1630 v3 CPU@3.7GHz (4 cores), 128GB DDR4 RAM, Xen/Linux versions 4.8*

# CPU usage




*Server: Intel Xeon E5-1630 v3 CPU@3.7GHz (4 cores), 128GB DDR4 RAM, Xen/Linux versions 4.8*

# Use cases

- Mobile-edge computing
  - Personal firewalls
  - Just-in-time services
- Lightweight computing

# Summary

- Virtualization is the backbone of today's computing infrastructure
- Hardware provides support for efficient virtualization
  - CPU, memory, IO
- VMs provide:
  - Strong isolation
  - And also lightweight: 2–200 ms instantiation times, memory footprint of 10s of MBs
- Achieved through:
  - Lightweight guests
  - Re-architected VM control plane