Virtualization and Live Snapshots

Lecture 12

Agenda

- What is Virtualization?
- Hardware Virtualization
- Operating System Virtualization
- Saving, Migrating, and Duplicating VM state
- Application: Low-cost Cloud Computing

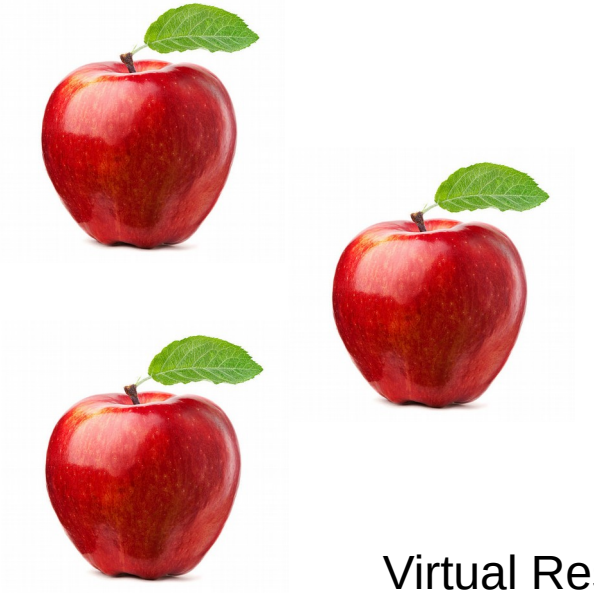
Operating Systems Are Amazing

- Complex system software that provide all the essential services and abstractions to applications
- Make hardware systems easy to use, secure, increase the utilization, portable, …
- Modern OSes like Linux are a feat of engineering
  - 10’s of millions of lines of code
  - Runs on wide range of devices (8-bit microcontrollers, raspberry pi, laptops, phones(android), every server connected to the internet
- But the success of operating systems also rests on key design principles that have been honed since the early days of computing
  - One key design principle is Virtualization

Virtualization

- Virtualization is a vital technique employed throughout the OS
- Given a resource, create the illusion of multiple virtual copies
- Users of the virtual resource (usually) cannot tell the difference

Resource

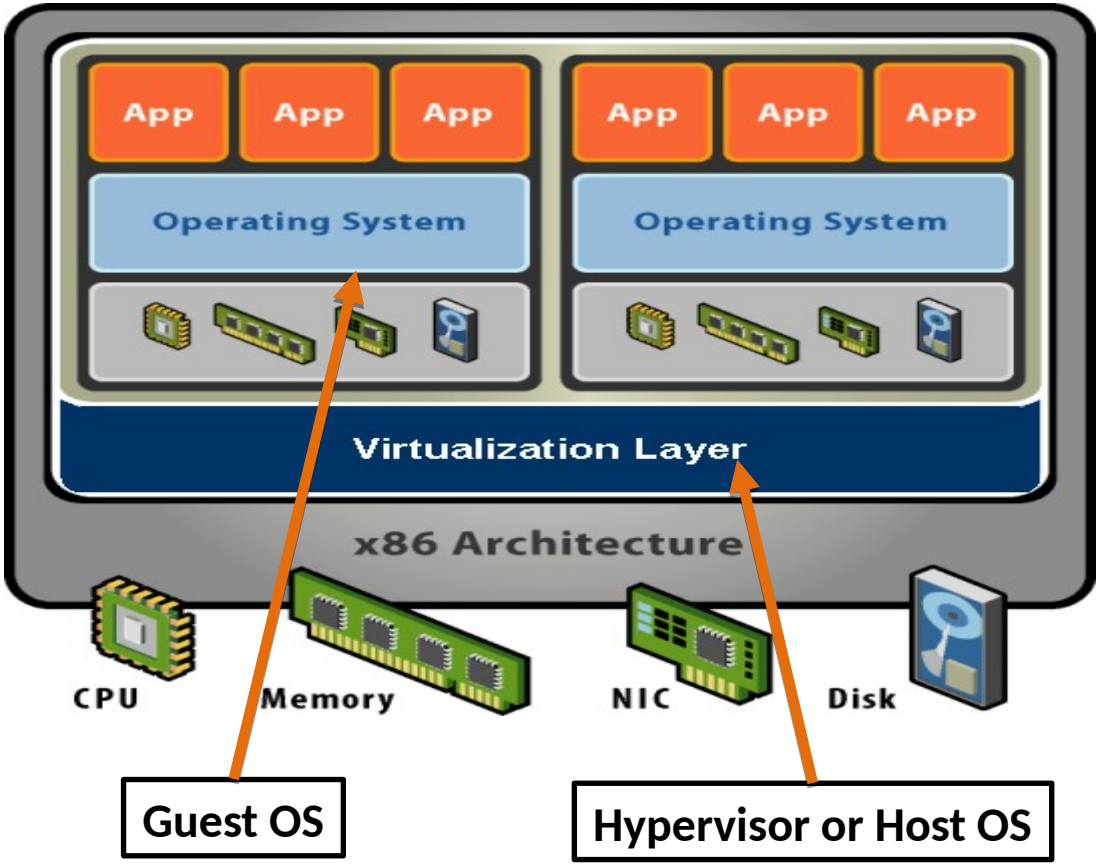


Virtual Resource

Virtualization In Operating Systems

- Virtualizing CPU enables us to run multiple concurrent processes
  - Time-division multiplexing and context switching
  - Provides multiplexing and isolation
- Similarly, virtualizing memory provides each process the illusion/abstraction of a large, contiguous, and isolated “virtual” memory
- Virtualizing a resource enables safe multiplexing

Virtual Machines: Virtualizing the Hardware



Vmware, Xen, KVM,..

- Software Abstraction
  - Behaves like hardware
  - Encapsulates all OS and application state
- Virtualization Layer
  - Extra level of indirection
  - Decouples hardware, OS
  - Enforces isolation
  - Multiplexes physical hardware across VMs

Hardware Virtualization History

- 1967: IBM System 360/ VM/370 fully virtualizable
- 1980s-1990s: ”Forgotten”. X86 had no support
- 1999: VMware. First x86 virtualization
- 2003: Xen. Paravirtualization for Linux. Used by Amazon EC2 public cloud
- 2006: Intel and AMD develop CPU extensions
- 2007: Linux Kernel Virtual Machine (KVM). Used in Google’s public cloud (and many others).

Why are Virtual Machines Interesting?

- Decouple OS from hardware
  - Run multiple OSes on a single machine
- Foundation of cloud computing
  - On-demand allocation of ”virtual servers”
  - Can launch a VM in seconds
- ”Easy” to implement interesting functionality:
  - Live migration of entire system from one machine to another (even across the internet)
  - Full-system (disk, memory, etc) snapshots/checkpoints
  - Time-travel debugging using record and replay
  - Primary-secondary replication for fault-tolerance

Hardware Virtualization 101

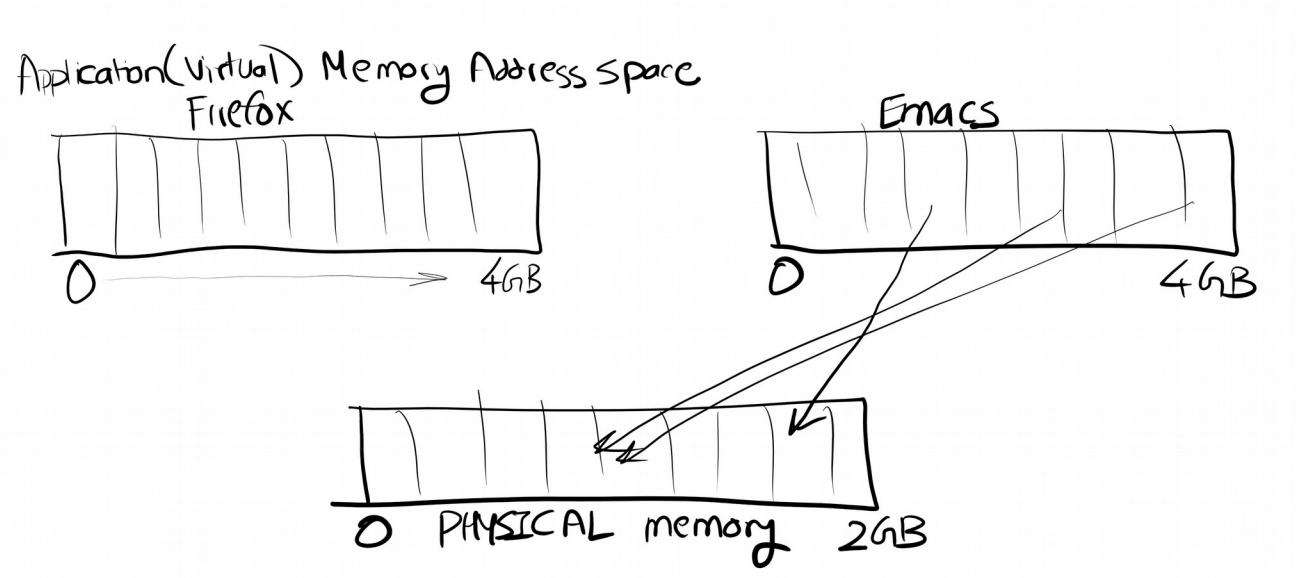
- Want to provide an illusion of a virtual CPU, virtual memory, virtual I/O devices to VM(s)
- Make virtual things appear and behave like physical things
- The hypervisor/VMM’s job is to safely virtualize resources
  - Performance of virtual resources should match physical resources!

Naïve Approach: Emulation

- Emulation: reproduce the behavior of hardware in software
- In our case: interpret and translate each CPU instruction and I/O operation
  - 10-100x performance overhead (QEMU)
  - Jslinux https://bellard.org/jslinux/
- Key requirement: Hypervisor must “get out of the way” as much as possible

Interlude: Virtual Memory

- Free applications from the shackles of physical memory
- Each process gets a large, contiguous virtual memory address space
- OS maps virtual addresses to physical addresses
- CPU har sical



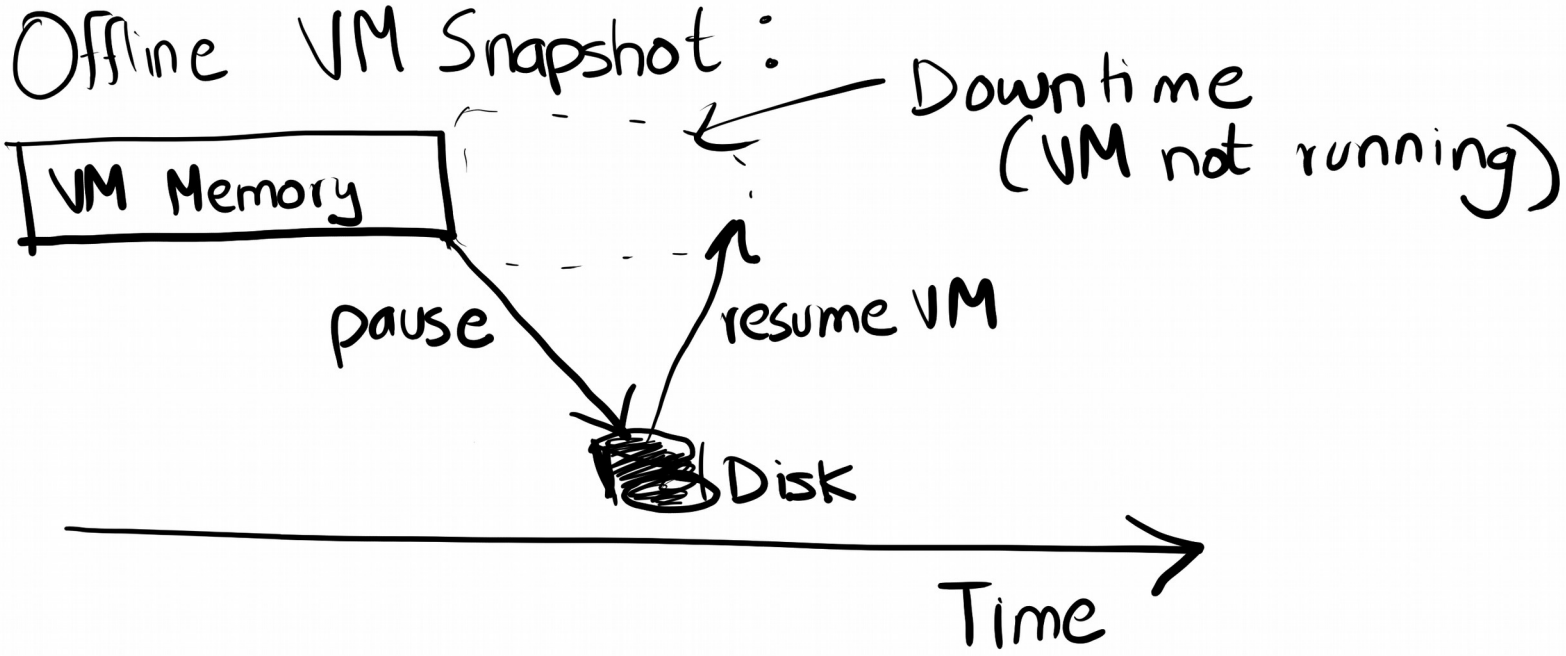
Hardware Virtualization Needs Another Layer Of Indirection

- Inside a VM: Guest Virtual Address → Guest Physical
- Hypervisor maintains: Guest Physical → Host Physical
- The actual address translation is again done in hardware
- But, this extra layer of indirection can be very useful to capture the memory state of a VM
  - Useful for taking snapshots, checkpoints that are “live”
    - Live => guest doesn’t stop executing
  - VM can migrate/move from one physical host to another
  - Debugging OS kernels, etc.

Live VM Memory Snapshots

VM Memory Snapshots

- We want to copy all VM memory to disk
- But, offline snapshots result in downtime for application

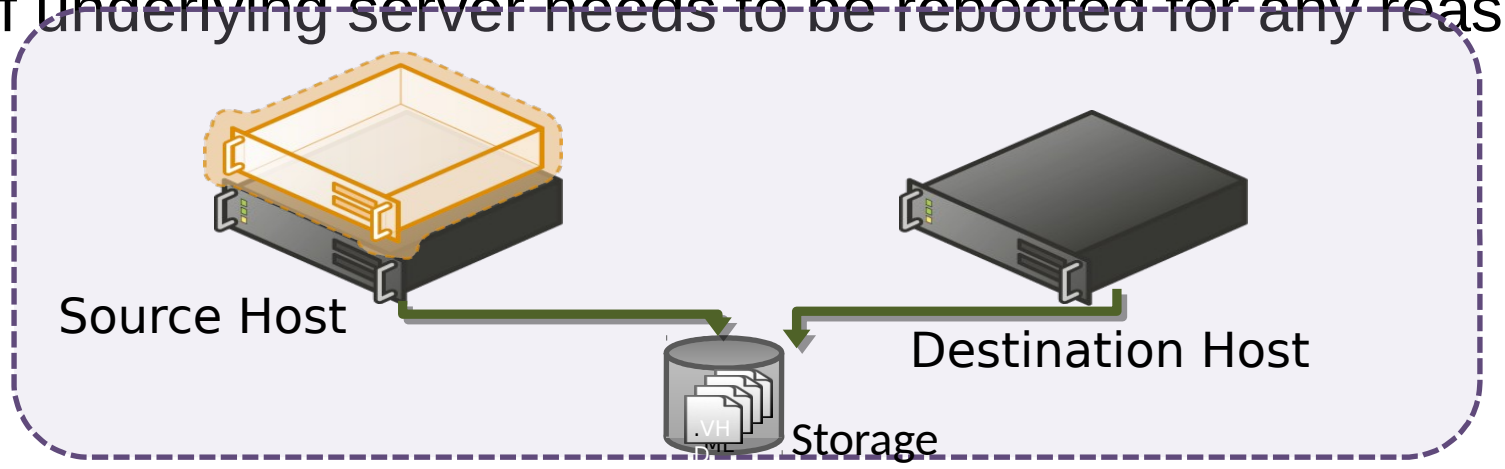


Live VM Snapshots

- Save memory “in the background” without stopping VM
- What can go wrong?
- Snapshot may not be consistent
- Live snapshot mechanism is also used in VM migration
  - Move VM from one server to another
  - Same as copying a snapshot and resuming on different server
- Also used for replication (Primary-Secondary)
  - Keep a secondary replica of the VM
  - Helps in fault tolerance!

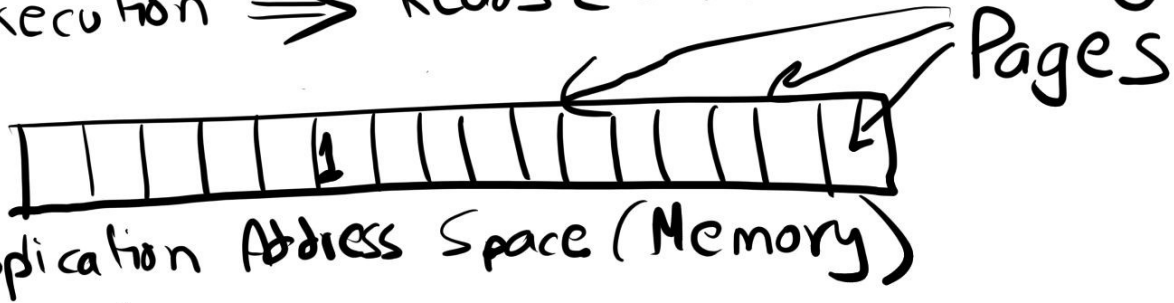
Migration

- Move VM between hosts while it is still running!
- No change in application behavior
- Useful for fault-tolerance, load balancing, etc.
  - If underlying server needs to be rebooted for any reason



VM Live Migration

- Applications execute inside VMs.
- Execution ⇒ Reads & Writes to memory locs.



X = 1;

All memory writes "Dirty" some page

Hypervisor identifies Dirty pages,

and sends them or stores them.

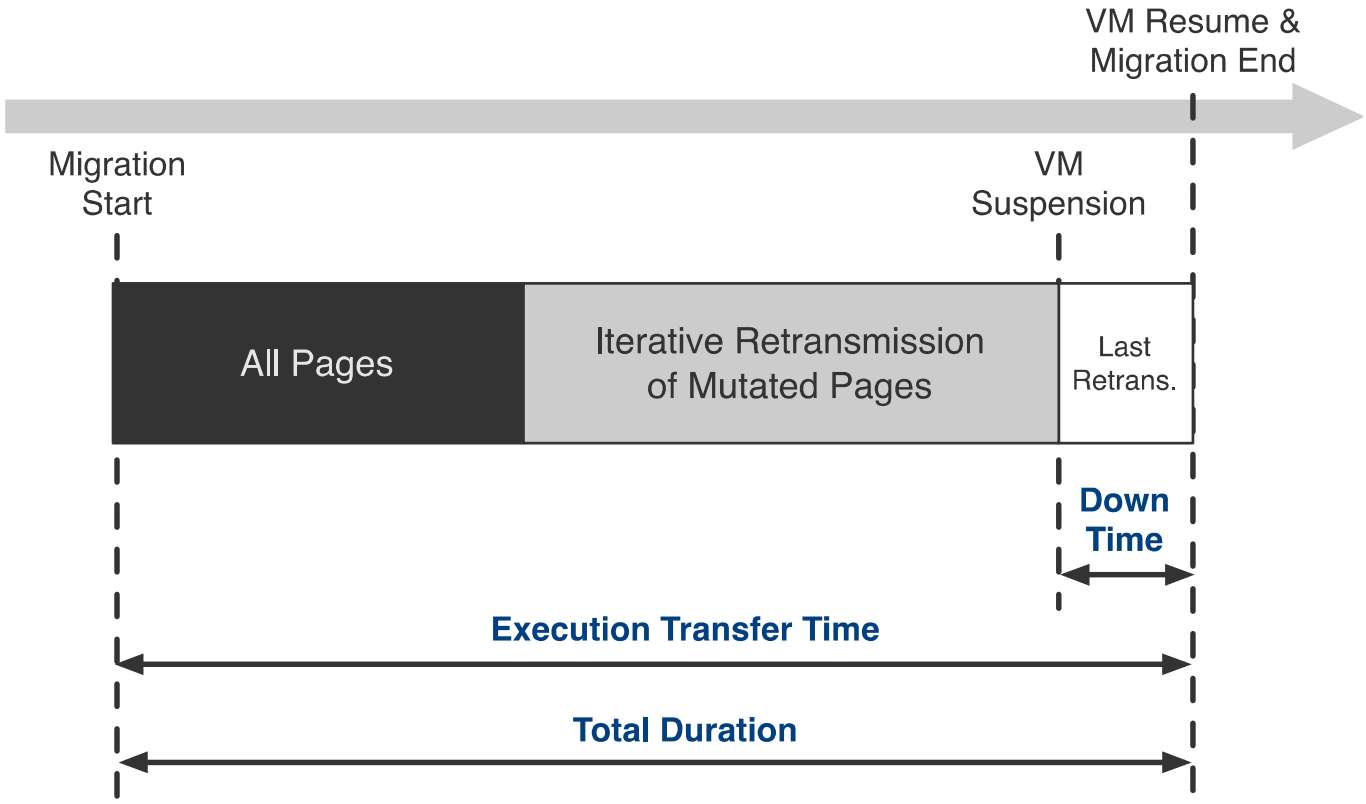
How To Find Which Pages Are Written By The VM?

- Recall that memory accesses are made directly by CPU
- BUT: extra layer of virtual memory indirection gives us a way!
- Recall: Guest Virtual → Guest Physical → Host Physical
- We want to be notified of all guest physical page writes
- Most common mechanism: Hypervisor marks all guest pages as write-only, and thus writes can be intercepted!
- Also on modern Intel CPUs: Page Monitoring Logging

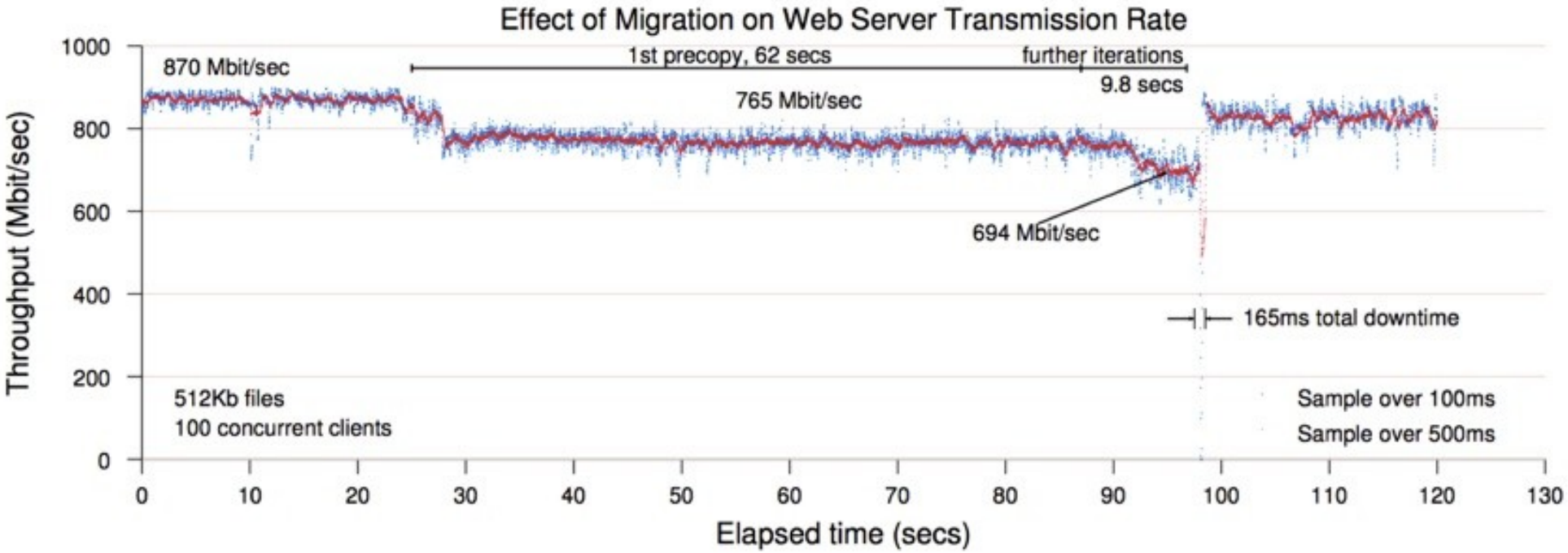
Pre-copy Iterative Snapshots

- While VM is running, copy all memory pages
- During the duration of the copy, some pages may have changed
- Identify those pages through page-tracking (dirty pages)
- In subsequent iteration, send only dirty pages
  - They are the “delta” between the two VM memory states
- Repeat a few times (~20)
- Stop when the number of dirty pages is tiny
- Stop the VM, and copy remaining dirty pages

Pre-copy Migration



Migration Performance

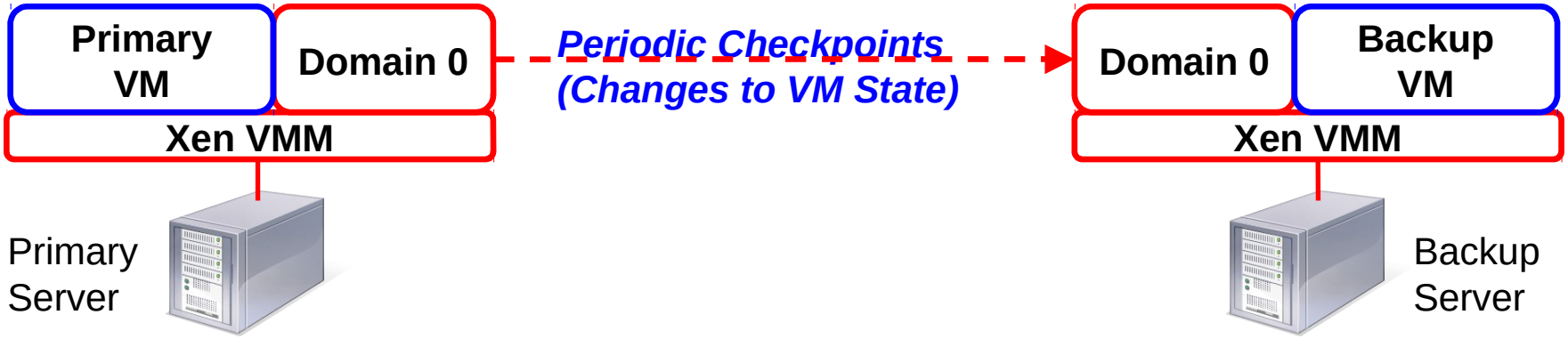


Remus [NSDI ‘08]

- System for High-availability for virtual machines
- Primary VM runs application
- Remus maintains an replica of this VM on another server
- Super useful for fault tolerance
  - If primary VM fails, just switch to secondary, and no one will notice!
- Builds on the live snapshot idea
- Main idea: take live snapshots continuously, so that we always have a very recent checkpoint to resume from

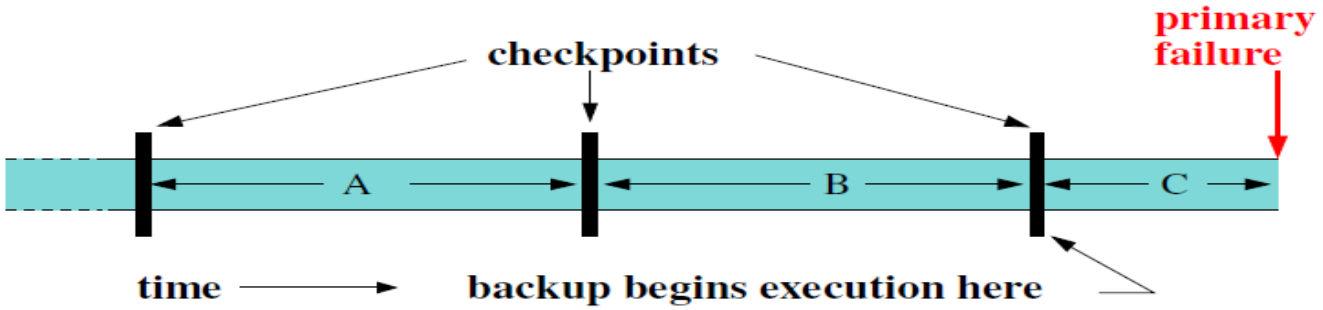
Remus Checkpoints

- Remus divides time into ***epochs*** (~25ms)
- Performs a ***checkpoint*** at the end of each epoch
  1. Suspend primary VM
  2. Copy all state changes to a buffer in ***Domain 0***
  3. Resume primary VM
  4. Send asynchronous message to backup containing state changes
  5. Backup VM applies state changes



[Ashraf Aboulnaga RemusDB]

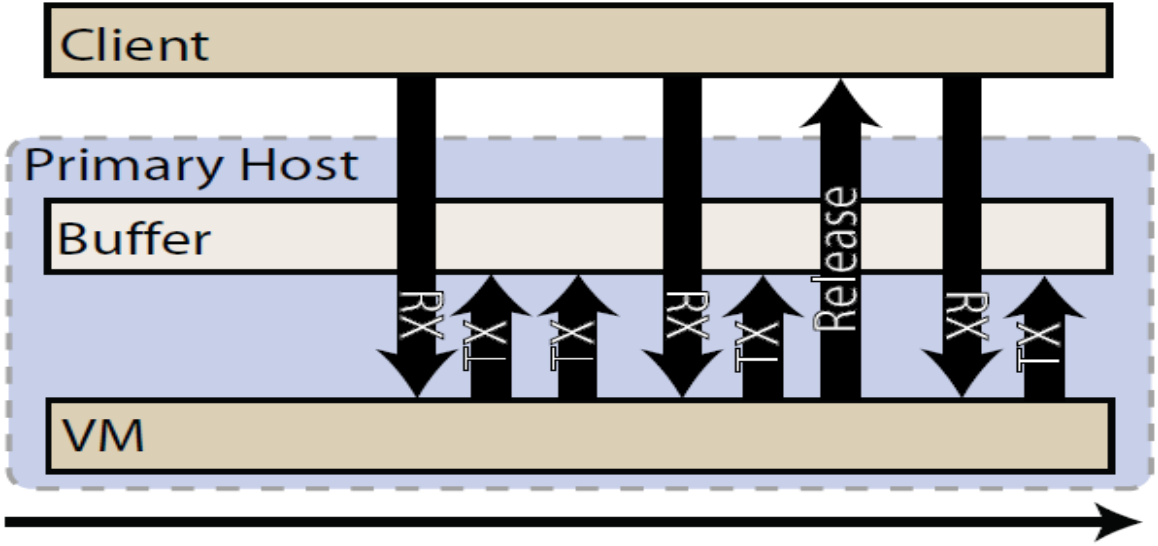
Remus Checkpoints



- After a failure, the backup resumes execution from the ***latest checkpoint***
  - Any work done by the primary during epoch C will be lost **(unsafe)**
- Remus provides a consistent view of execution to clients
  - Any network packets sent during an epoch are ***buffered*** until the next checkpoint
  - Guarantees that a client will see results only if they are based on ***safe*** execution
  - Same principle is also applied to disk writes

[Ashraf Aboulnaga RemusDB]

Outbound packet buffering



Bounded Time VM Live Migration

- Live migration’s pre-copy phase can take a long time (few minutes)
- Remus requires 2x the number of servers
- Bounded-time VM live migration combines these two techniques:
  - Migrations finish fast, within pre-specified time
  - Multiple VM snapshots stored on shared backup disk

Record-Replay of VMs

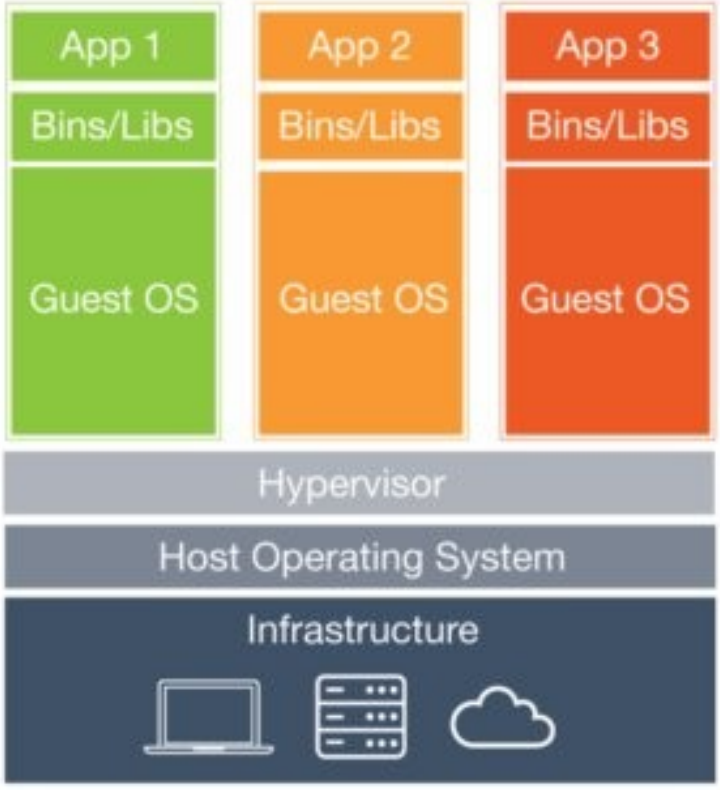
- Another way for replicating a VM is to use the classic state machine approach
- Record all external events that a VM faces, and replay them at a later time (or somewhere else)
- Hypervisor can record precisely when an event occurs, and can inject it into the VM at a precise instruction
- This uses CPU performance counters (instruction counting, branch counting), and debug registers

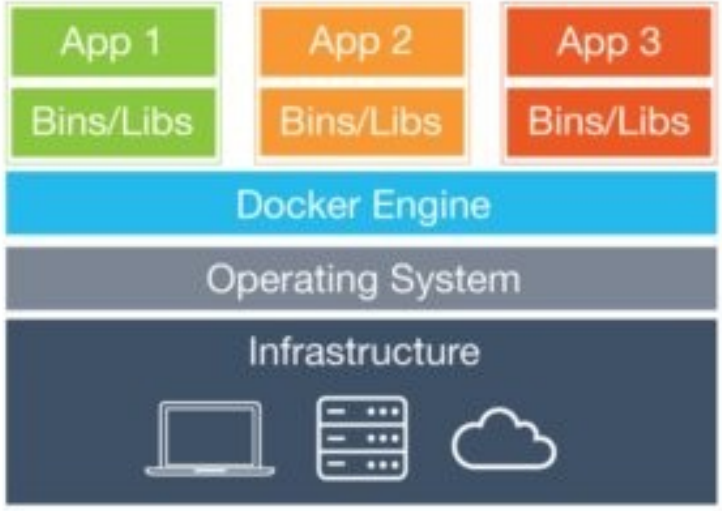
OS Virtualization

OS Virtualization

- Virtualize the OS for each application
- Aka “Containers”
- Each container gets an own virtual copy of the OS
  - Ofcourse, there’s only one “real” OS
- Containers allow “sandboxing” of applications
  - Applications are isolated and unaware of each other
- OS virtualization is “light weight” compared to full HW virt
- No need to double-virtualize memory, CPU, I/O devices
- Containers

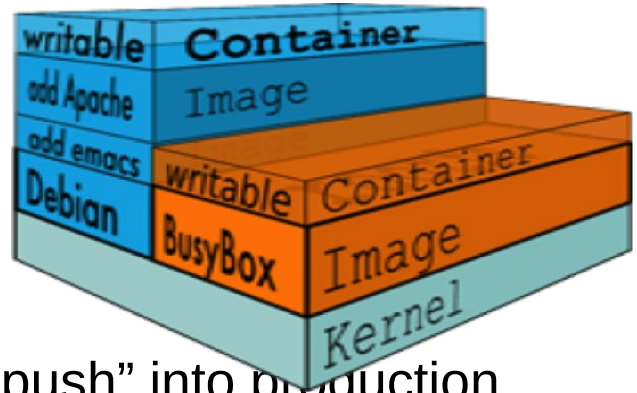
HW vs. OS Virtualization





Docker

- Containers + layered file system + image repository
- Copy-on-write file system allows images to be composed layer-wise
  - Base layer: Debian
  - Layer 2: Essentials (Emacs)
  - Layer 3: Apache web server
- Common use-case: CICD
  - Continuous Integration and Deployment
  - Create docker container in dev environment, “push” into production



- Docker-files allow capturing dependency information between files, libraries, and packages
  - Similar to how Makefiles capture dependency between files

END