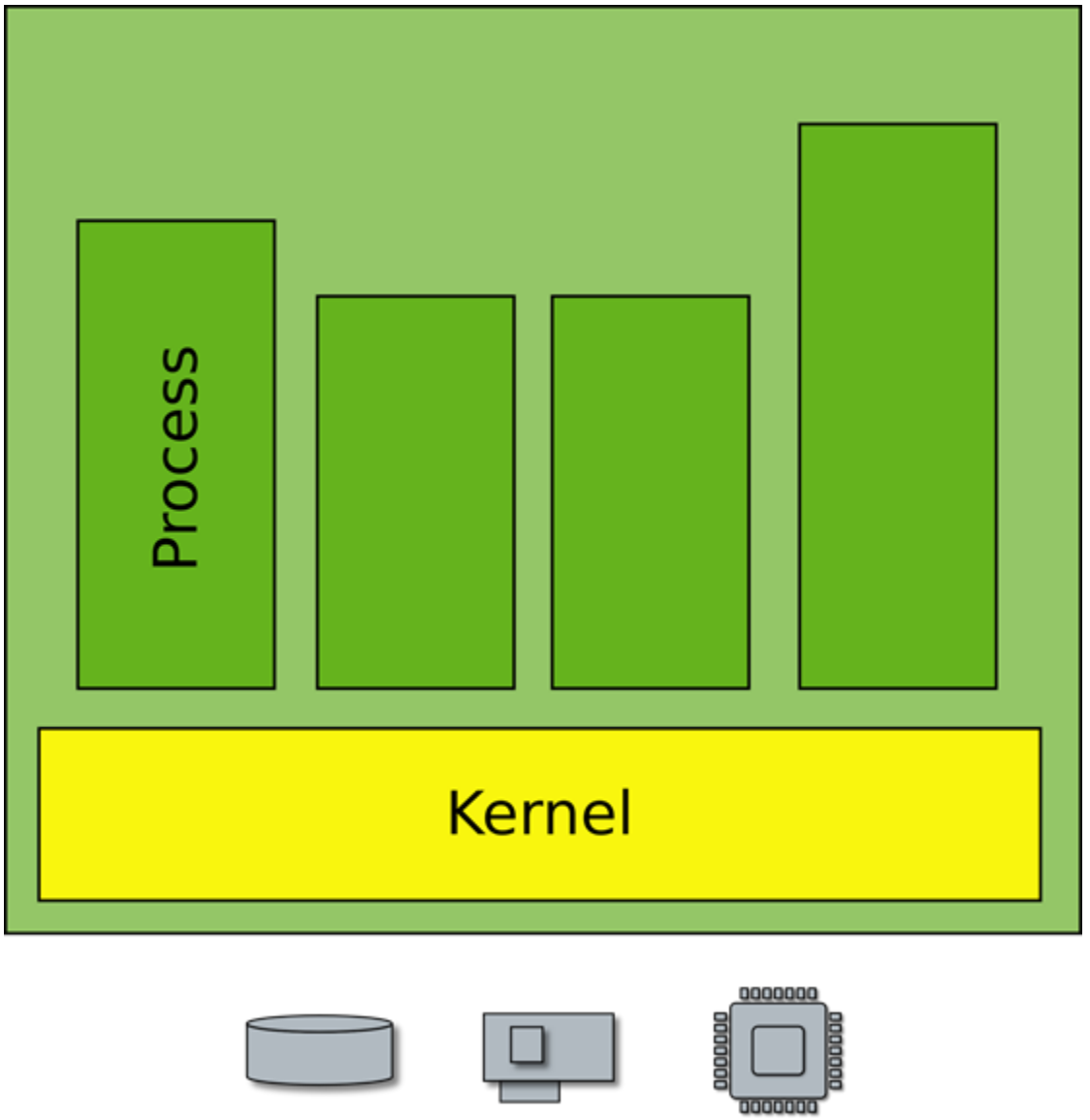
CS6465: Advanced Operating System Implementation

Lecture 07: Virtualization

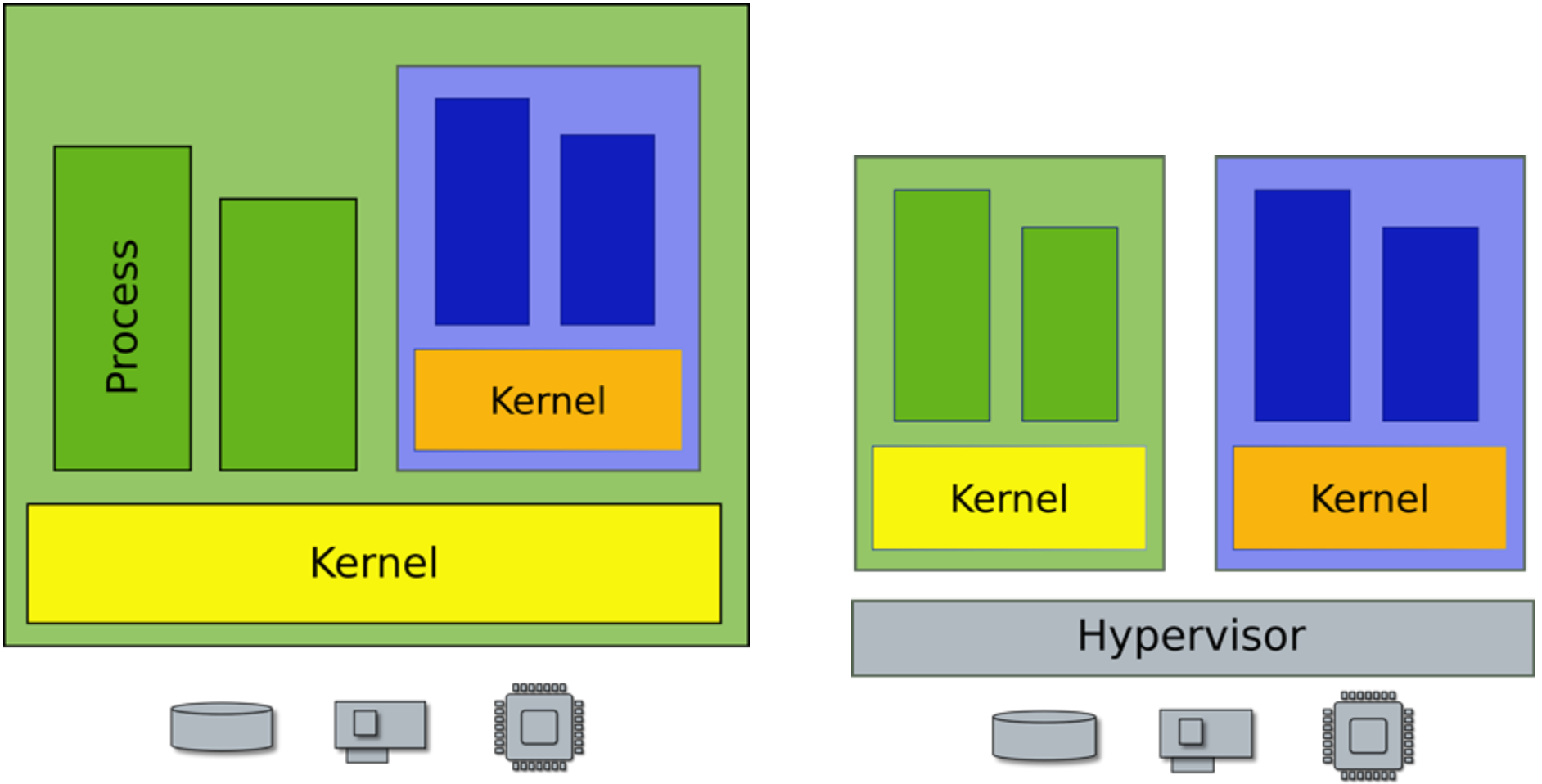
Anton Burtsev,

October, 2024

Traditional operating system



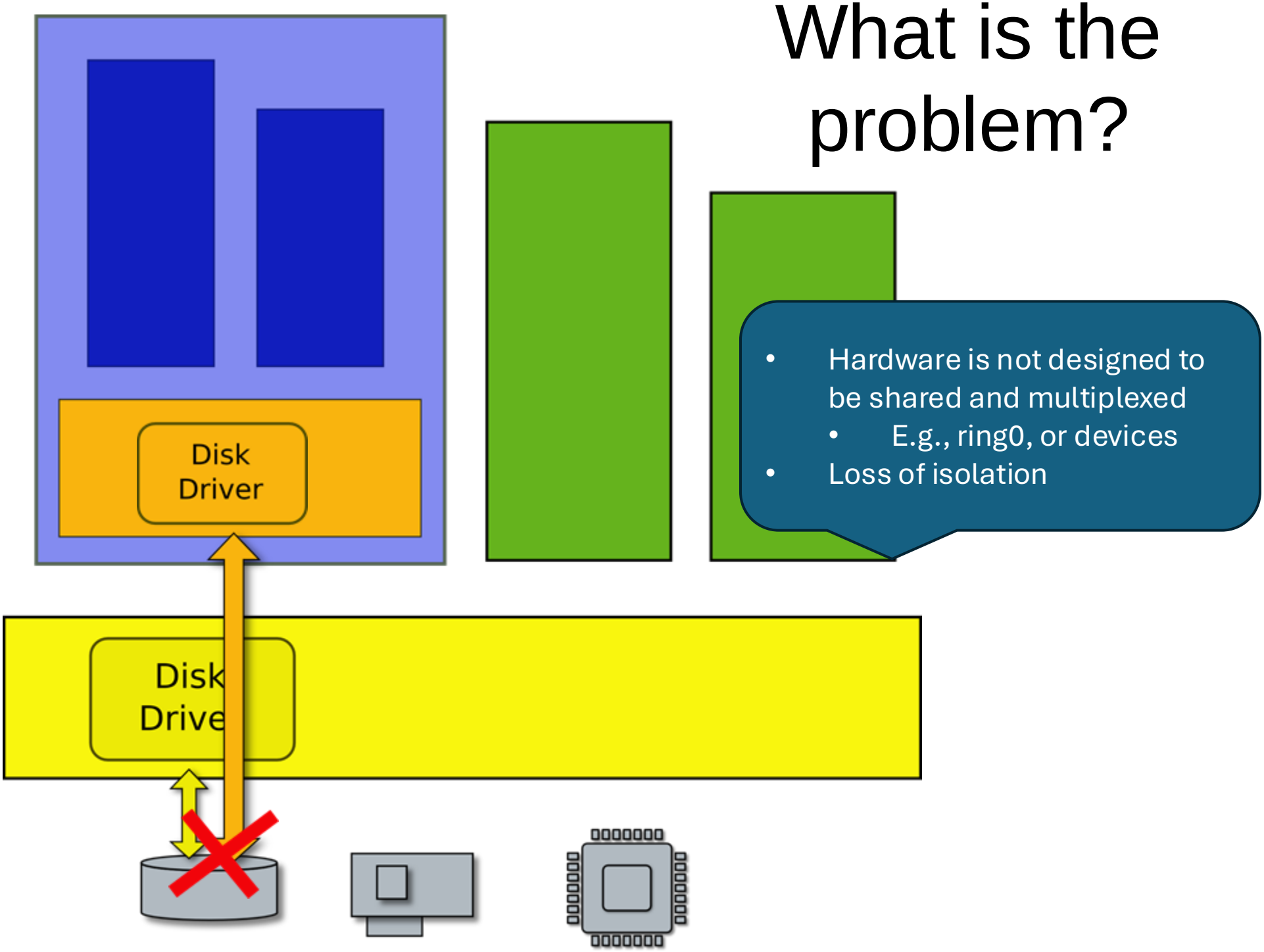
Virtual machines



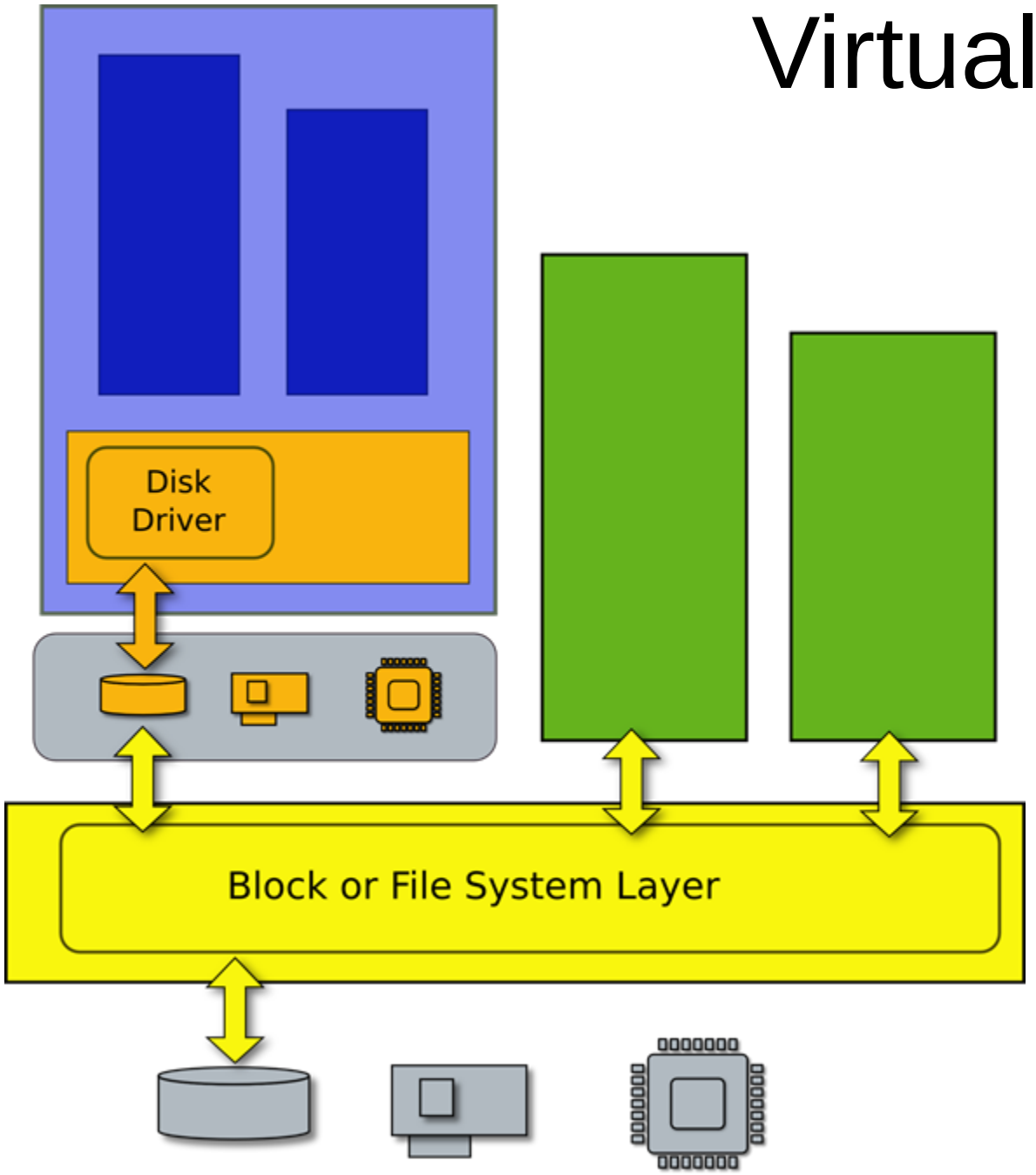
A bit of history

- Virtual machines were popular in 60s-70s
- Share resources of mainframe computers [Goldberg 1974]
- Run multiple single-user operating systems
- Interest is lost by 80s-90s
- Development of multi-user OS
- Rapid drop in hardware cost
- Hardware support for virtualizaiton is lost

What is the problem?



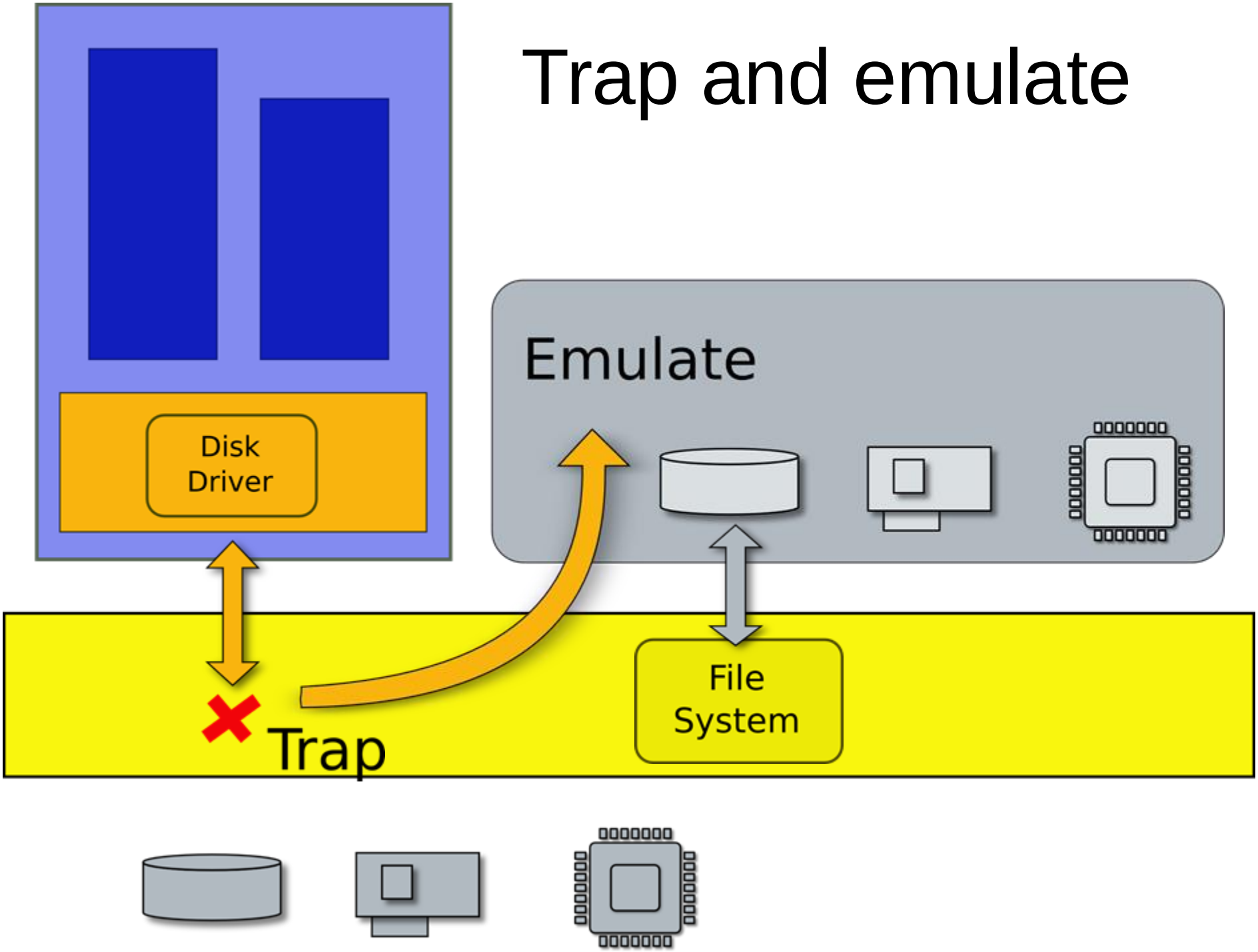
Virtual machine



Efficient duplicate of a real machine

- Compatibility
- Performance
- Isolation

Trap and emulate



What needs to be emulated?

- CPU and memory
  - Register state
  - Memory state
    - Memory management unit
    - Page tables, segments
- Platform
  - Interrupt controller, timer, buses
    - BIOS
- Peripheral devices
  - Disk, network interface, serial line

x86 is not virtualizable

- Some instructions (*sensitive*) read or update the state of virtual machine and don't trap (*non-privileged*)
- 17 sensitive, non-privileged instructions [Robin et al. 2000]

x86 is not virtualizable (II)

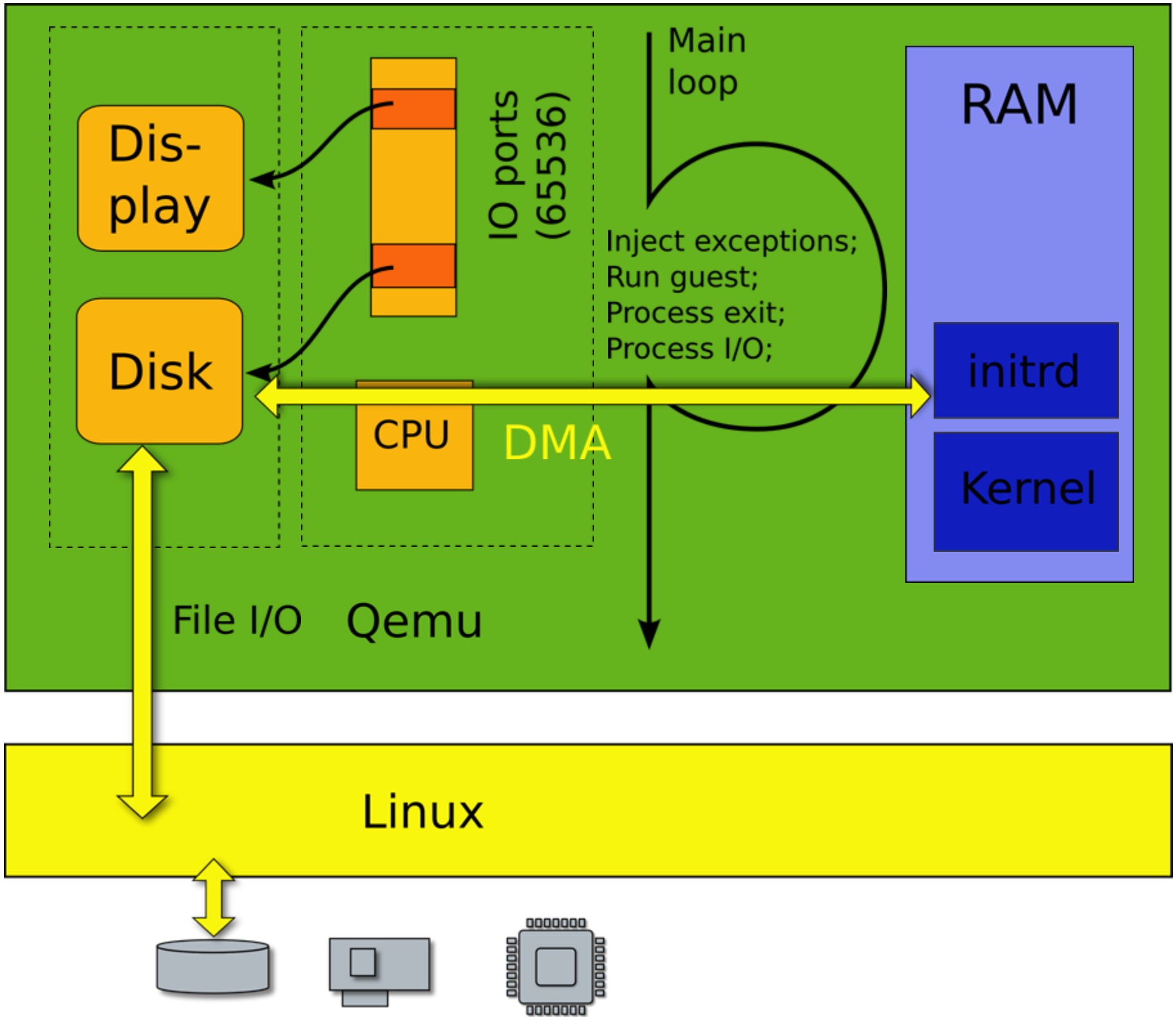
Group	Instructions
Access to interrupt flag	`pushf, popf, iret`
Visibility into segment descriptors	`lar, verr, verw, lsl`
Segment manipulation instructions	`pop <seg>, push <seg>, mov <seg>`
Read-only access to privileged state	`sgdt, sldt, sidt, smsw`
Interrupt and gate instructions	`fcall, longjump, retfar, str, int <n>`

- Examples
- 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

Solution space

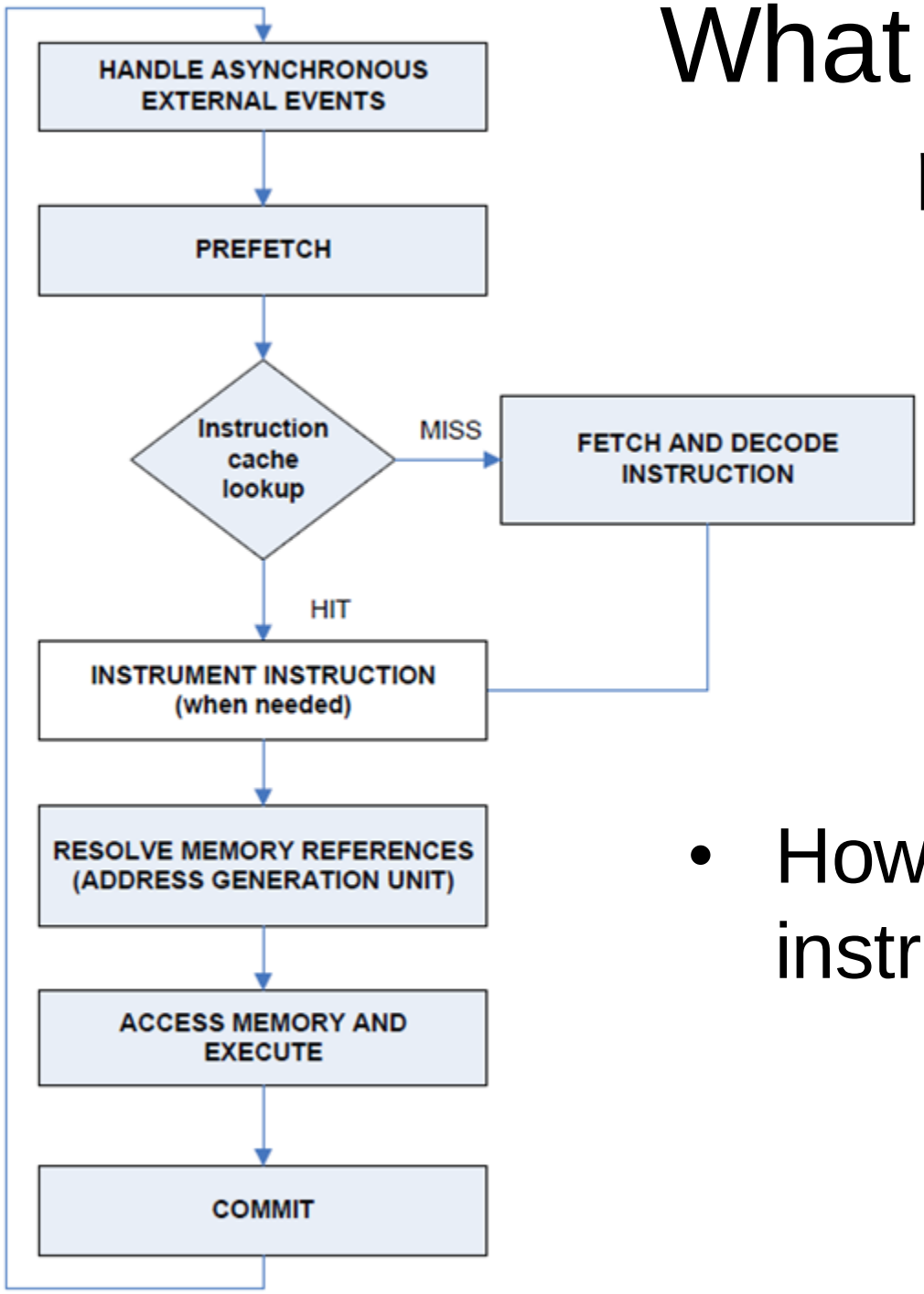
- Parse the instruction stream and detect all sensitive instructions dynamically
  - Interpretation (BOCHS, JSLinux)
  - Binary translation (VMWare, QEMU)
- Change the operating system
  - Paravirtualization (Xen, L4, Denali, Hyper-V)
- Make all sensitive instructions privileged!
  - Hardware supported virtualization (Xen, KVM, VMWare)
  - Intel VT-x, AMD SVM

Basic blocks of a virtual machine monitor: QEMU example


Main
loop
RAM
Dis-
play
IO ports
(65536)
Inject exceptions;
Run guest;
Process exit;
Process I/O;
Disk
initrd
CPU
DMA
Kernel
File I/O
Qemu
Linux

Interpreted execution:

BOCHS, JSLinux



What does it mean to run guest?

- Bochs internal emulation loop
- Similar to non-pipelined CPU like 8086

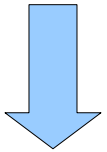
- How many cycles per instruction?

Binary translation: VMWare/QEMU

```
int isPrime(int a) {
  for (int i = 2; i < a; i++) {
    if (a % i == 0) return 0;
  }
  return 1;
}
```

```
isPrime:  mov    %ecx, %edi ; %ecx = %edi (a)
          mov    %esi, $2   ; i = 2
          cmp    %esi, %ecx ; is i >= a?
          jge    prime      ; jump if yes
nexti:    mov    %eax, %ecx ; set %eax = a
          cdq               ; sign-extend
          idiv   %esi       ; a % i
          test   %edx, %edx ; is remainder zero?
          jz     notPrime   ; jump if yes
          inc    %esi       ; i++
          cmp    %esi, %ecx ; is i >= a?
          jl     nexti      ; jump if no
prime:    mov    %eax, $1   ; return value in %eax
          ret
notPrime: xor    %eax, %eax ; %eax = 0
          ret
```

```
isPrime:   mov    %ecx, %edi ; %ecx = %edi (a)
           mov    %esi, $2   ; i = 2
           cmp    %esi, %ecx ; is i >= a?
           jge    prime      ; jump if yes
nexti:     mov    %eax, %ecx ; set %eax = a
           cdq               ; sign-extend
           idiv   %esi       ; a % i
           test   %edx, %edx ; is remainder zero?
           jz     notPrime   ; jump if yes
           inc    %esi       ; i++
           cmp    %esi, %ecx ; is i >= a?
           jl     nexti      ; jump if no
prime:     mov    %eax, $1   ; return value in %eax
           ret
notPrime:  xor    %eax, %eax ; %eax = 0
           ret
```



```
isPrime':    mov %ecx, %edi    ; IDENT
             mov %esi, $2
             cmp %esi, %ecx
             jge [takenAddr]   ; JCC
             jmp [fallthrAddr]
```

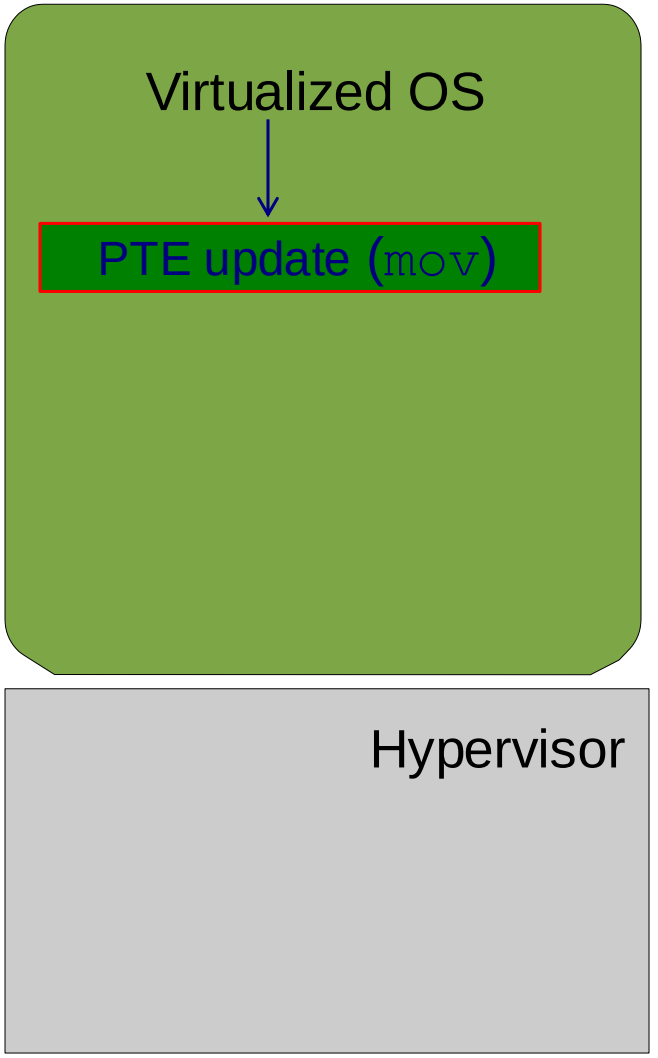
```
isPrime':  *mov   %ecx, %edi    ; IDENT
            mov   %esi, $2
            cmp   %esi, %ecx
            jge   [takenAddr]   ; JCC
                                ; fall-thru into next CCF
nexti':    *mov   %eax, %ecx    ; IDENT
            cdq
            idiv  %esi
            test  %edx, %edx
            jz    notPrime'     ; JCC
                                ; fall-thru into next CCF
           *inc   %esi          ; IDENT
            cmp   %esi, %ecx
            jl    nexti'        ; JCC
            jmp   [fallthrAddr3]

notPrime': *xor   %eax, %eax    ; IDENT
            pop   %r11          ; RET
            mov   %gs:0xff39eb8(%rip), %rcx  ; spill %rcx
            movzx %ecx, %r11b
            jmp   %gs:0xfc7dde0(8*%rcx)
```

Paravirtualization: Xen

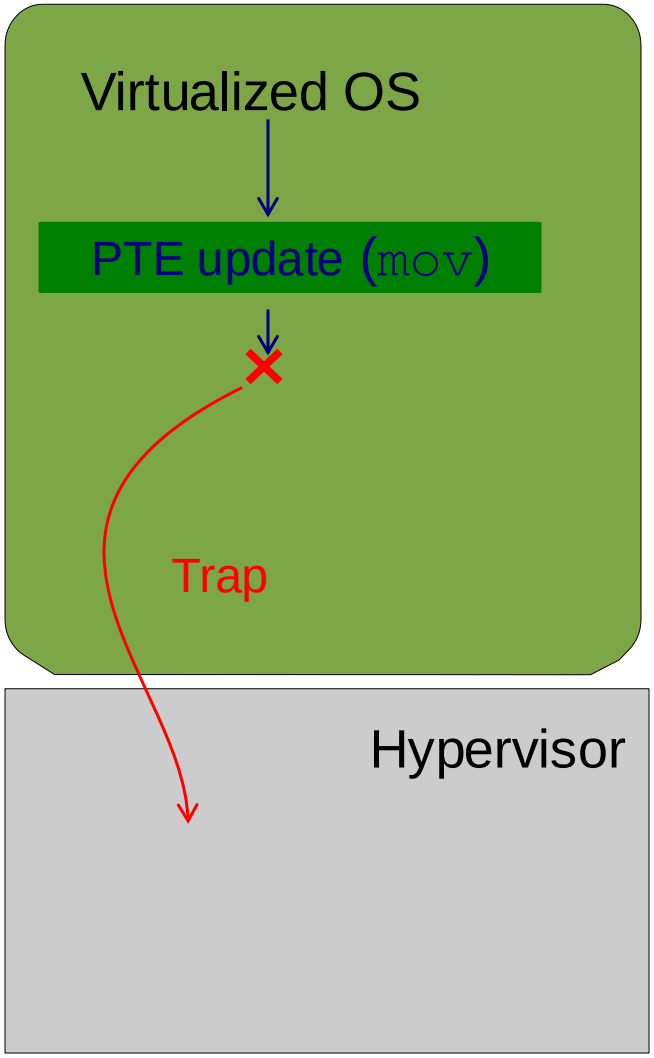
Full virtualization

- Complete illusion of physical hardware
- Trap _all_ sensitive instructions
- Example: page table update



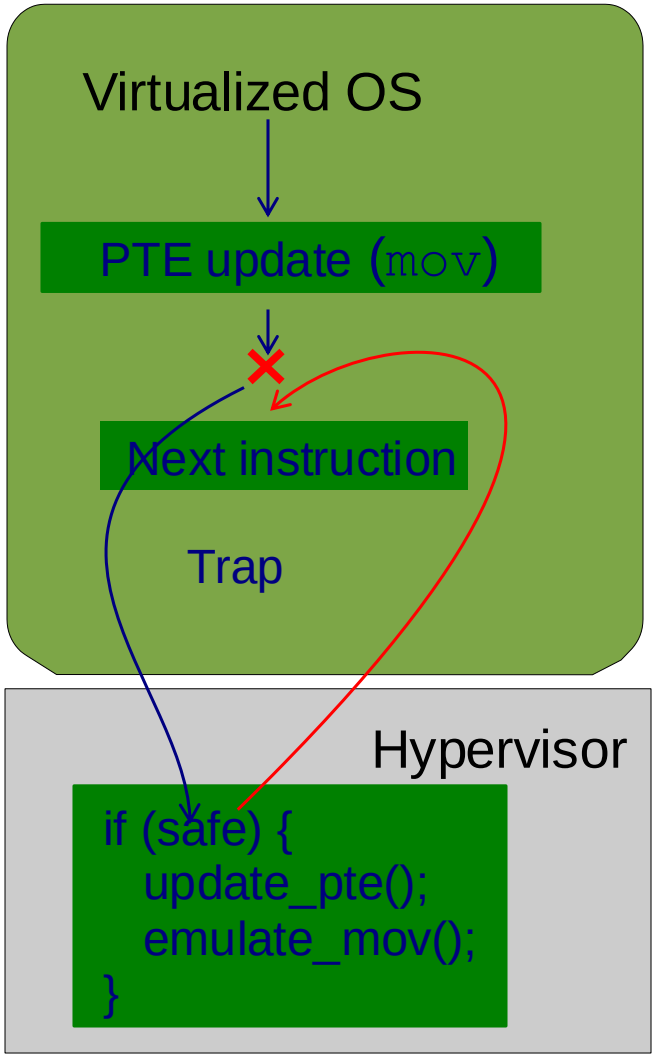
Full virtualization

- Complete illusion of physical hardware
- Trap _all_ sensitive instructions
- Example: page table update



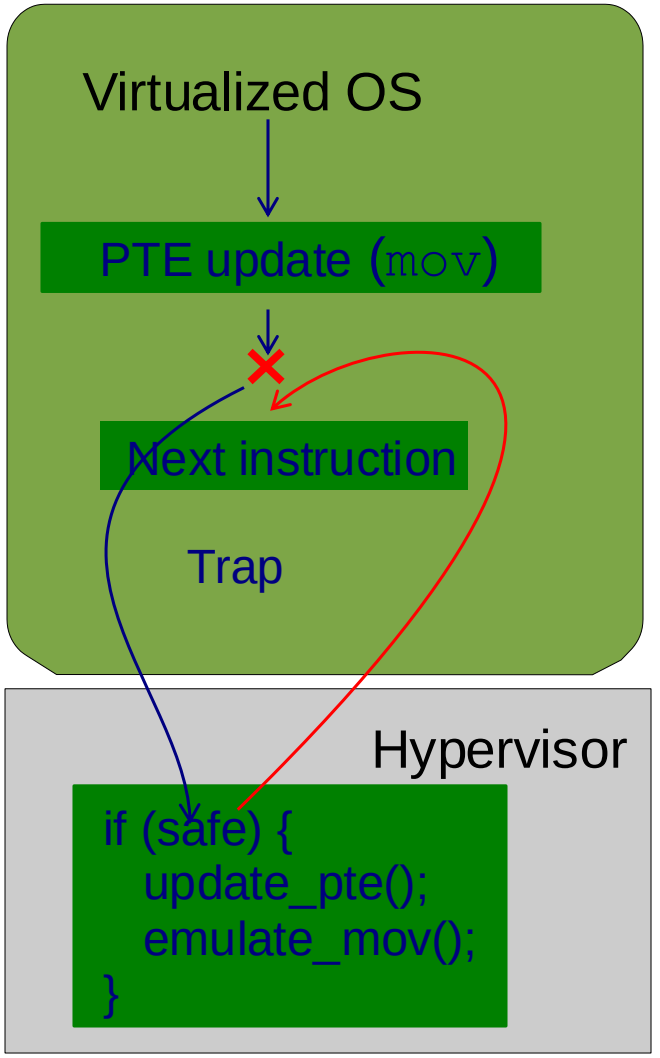
Full virtualization

- Complete illusion of physical hardware
- Trap _all_ sensitive instructions
- Example: page table update



Performance problems

- Traps are slow
- Binary translation is faster
- For some events

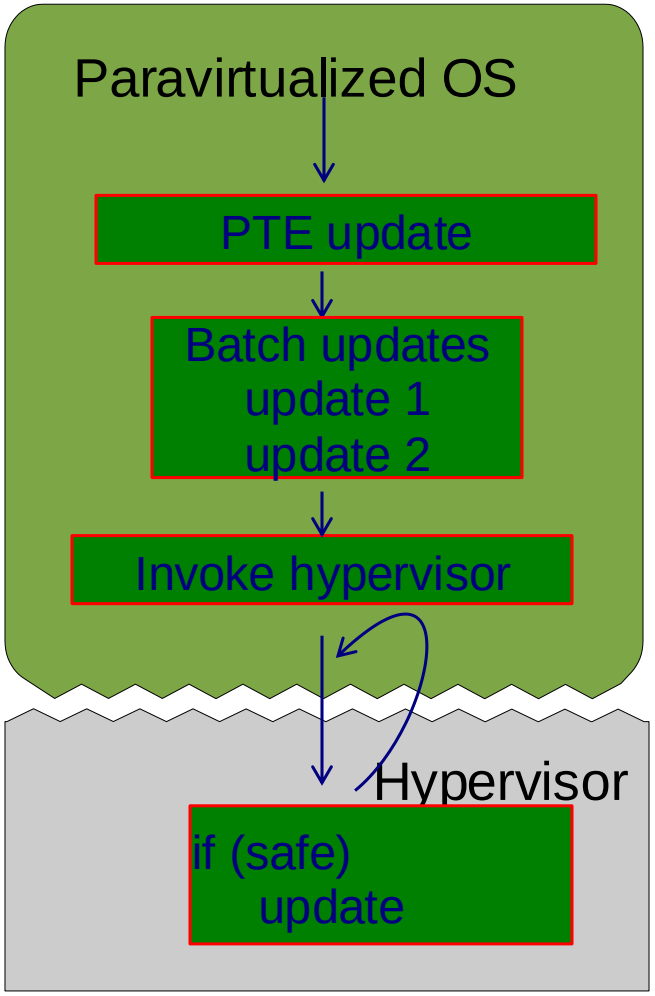


Paravirtualization

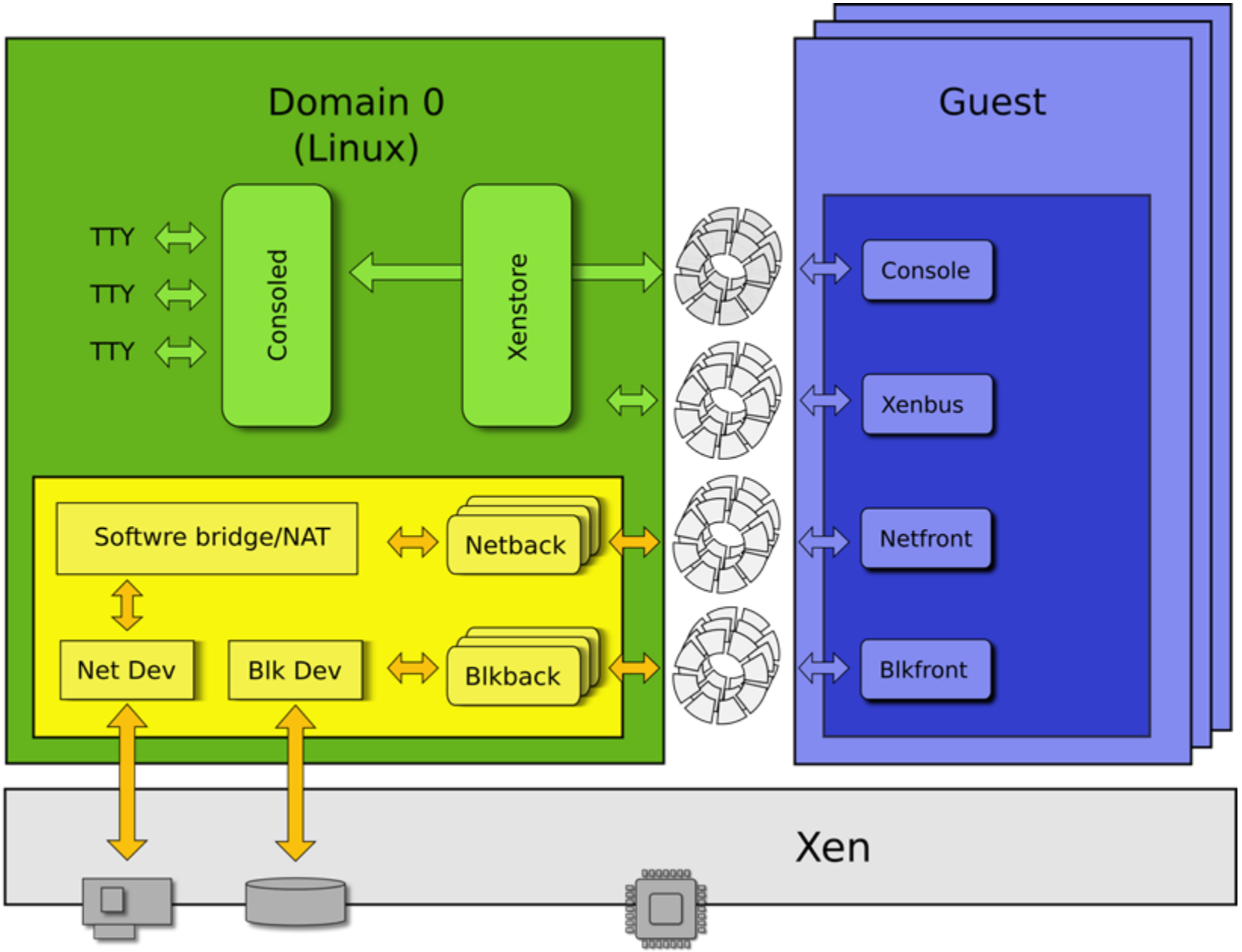
- No illusion of hardware
- Instead: paravirtualized interface
- Explicit hypervisor calls to update sensitive state
  - Page tables, interrupt flag

- But Guest OS needs porting
- Applications run natively in Ring 3

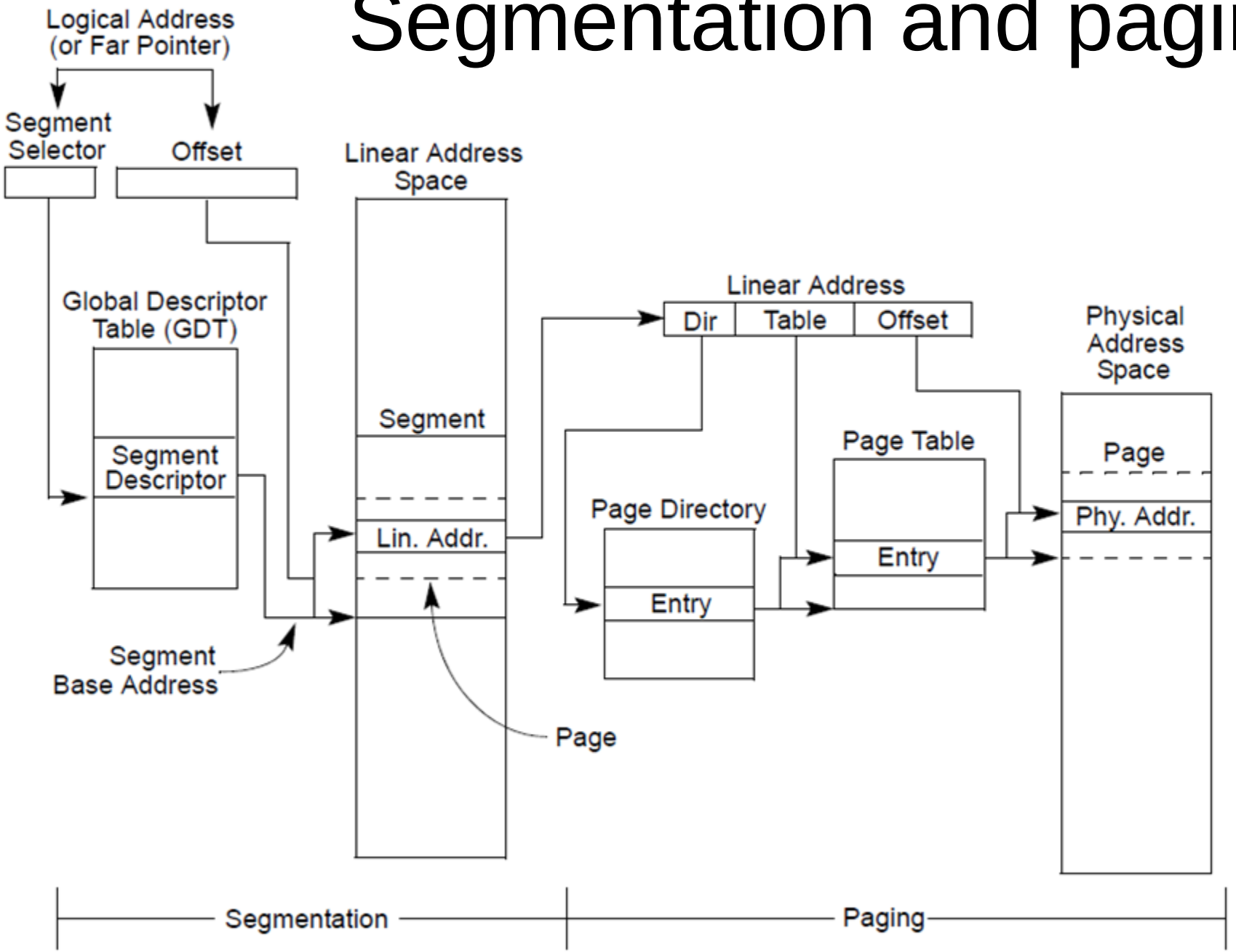
Paravirtualization



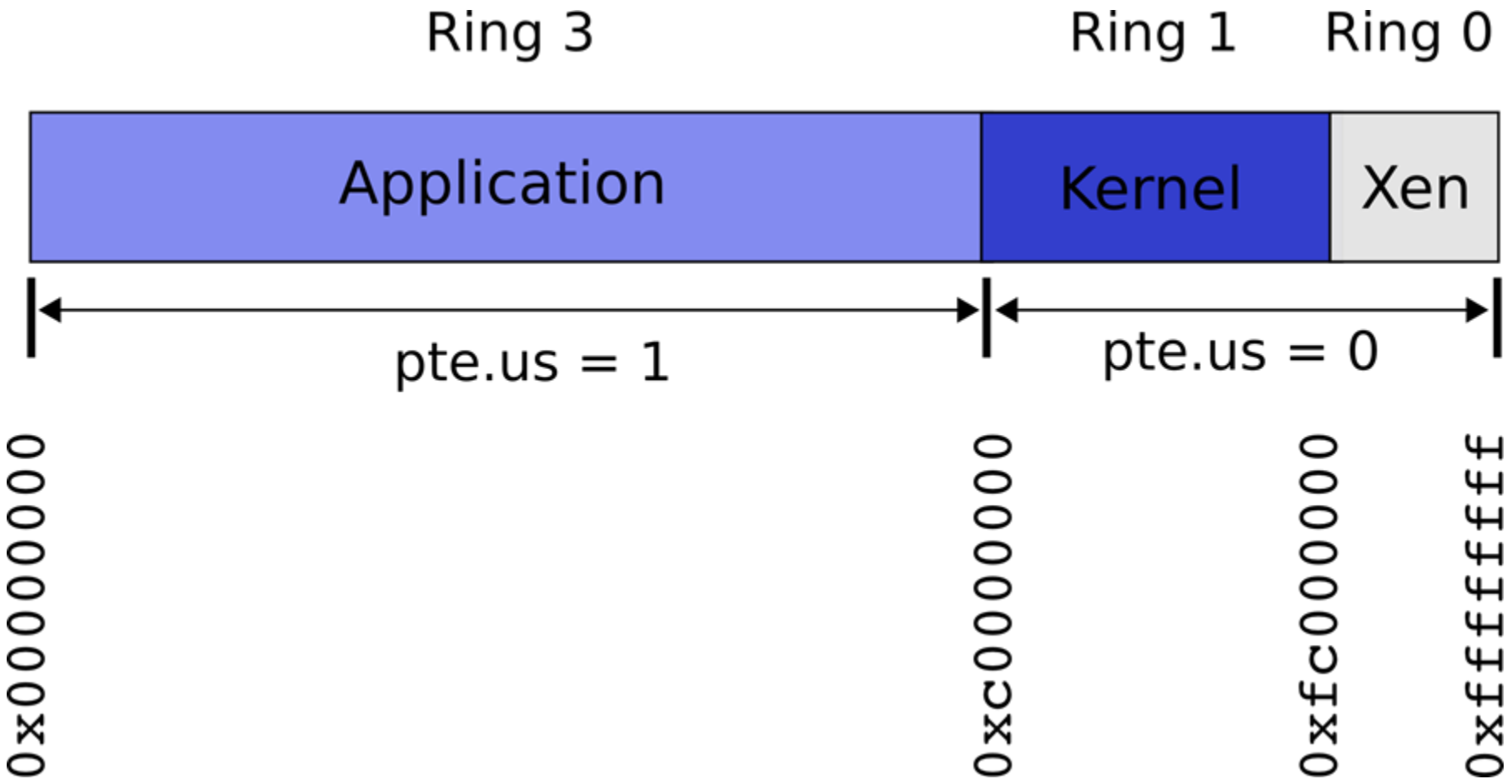
Xen


Domain 0
(Linux)
TTY
TTY
TTY
Consoled
Xenstore
Softwre bridge/NAT
Netback
Net Dev
Blk Dev
Blkback
Guest
Console
Xenbus
Netfront
Blkfront
Xen

Segmentation and paging

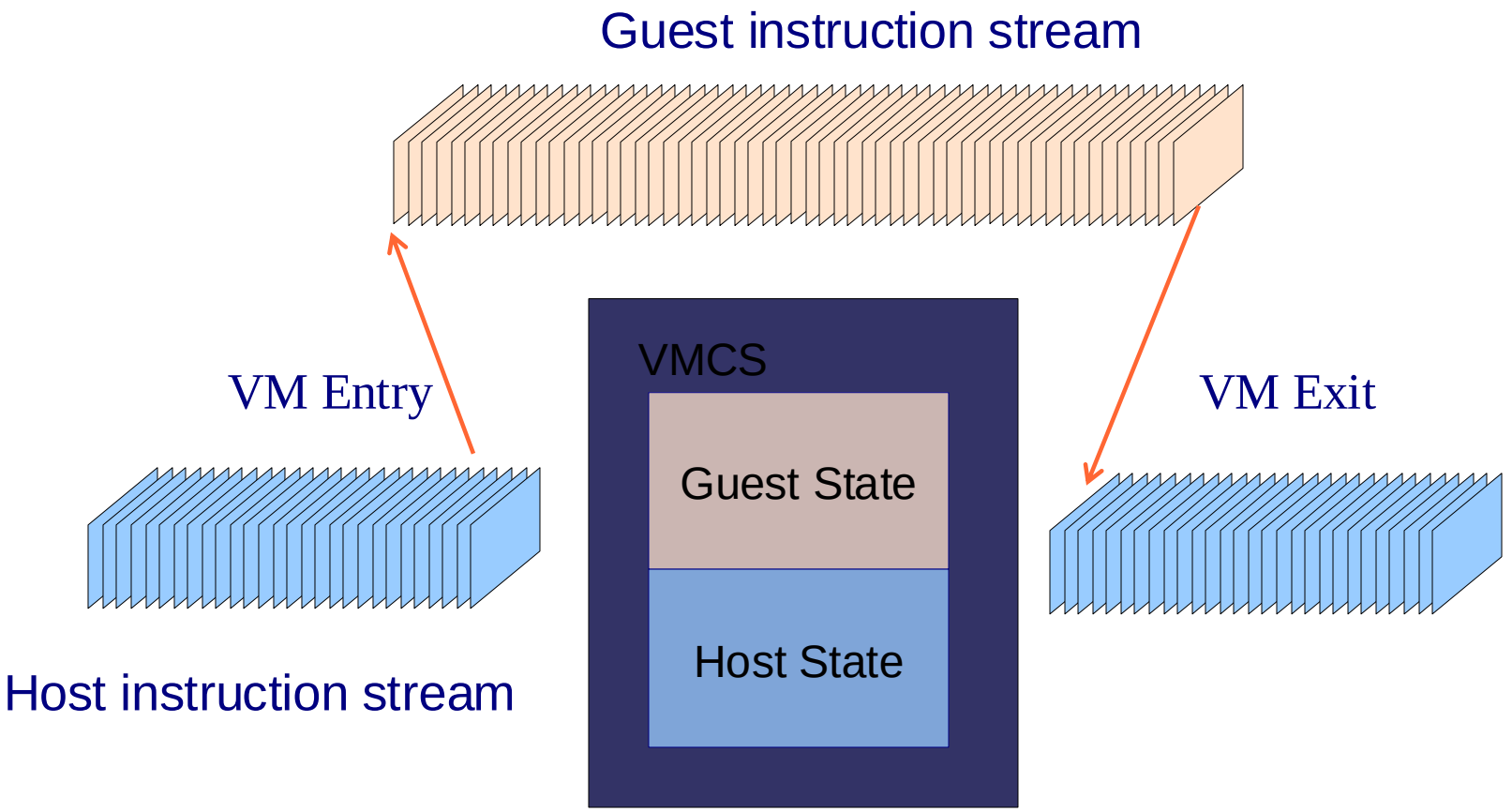


Hypervisor protection



Hardware support for virtualization: KVM, Modern Xen, Modern Qemu

Basic idea


Guest instruction stream
VM Entry
VMCS
Guest State
Host State
VM Exit
Host instruction stream

New mode of operation:VMX root

- VMX root operation
  - 4 privilege levels
- VMX non-root operation
  - 4 privilege levels as well, but unable to invoke VMX root instructions
  - Guest runs until it performs exception causing it to exit
- Rich set of exit events
  - Guest state and exit reason are stored in VMCS

Virtual machine control structure (VMCS)

- Guest State
  - Loaded on entries
  - Saved on exits
- Host State
  - Saved on entries
  - Loaded on exits
- Control fields
  - Execution control, exits control, entries control

Guest state

- Register state
- Non-register state
- Activity state:
  - active
  - inactive (HLT, Shutdown, wait for Startup IPI interprocessor interrupt))
- Interruptibility state

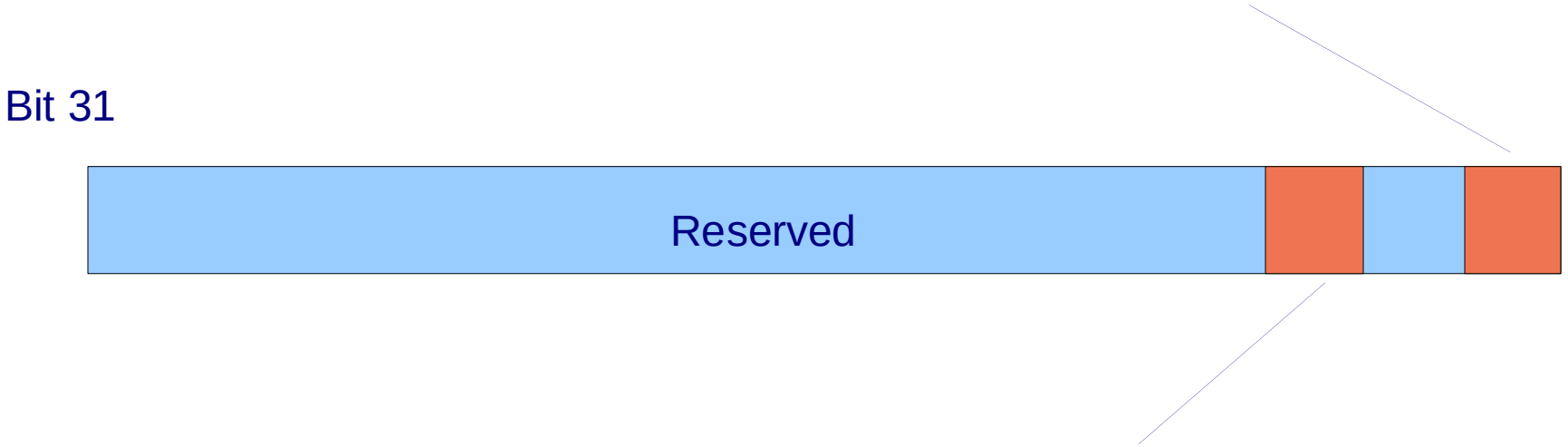
Host state

- Only register state
  - ALU registers,
- also:
  - Base page table address (CR3)
  - Segment selectors
  - Global descriptors table
  - Interrupt descriptors table

VM-execution controls

(asynchronous events control)

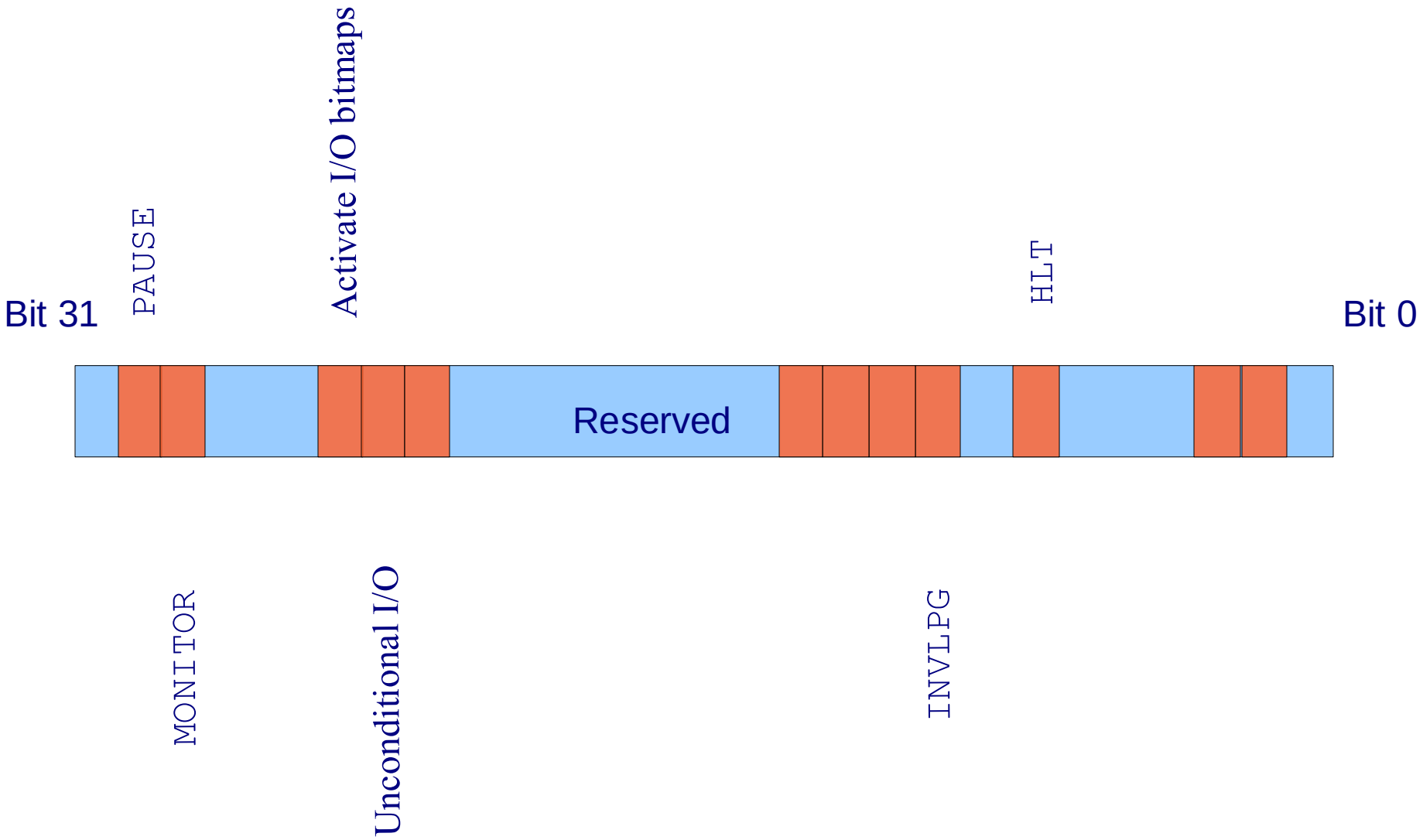
External interrupts (maskable or IRQs) cause exits(yes/no)
If not, then they delivered through guest
IDT



NMI cause exits (yes/no)
If not, then they are delivered normally through guest IDT (descriptor 2)

VM-execution controls

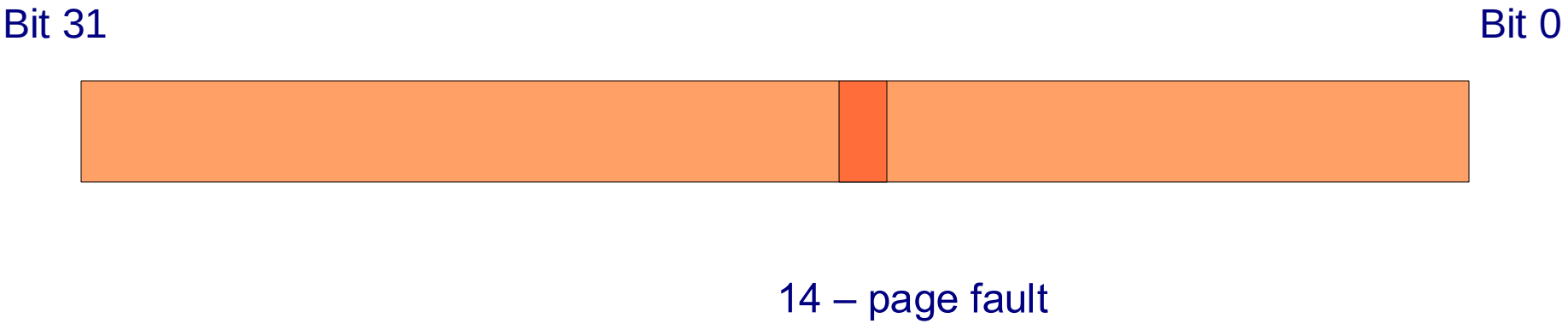
(synchronous events control, not all reasons are shown)



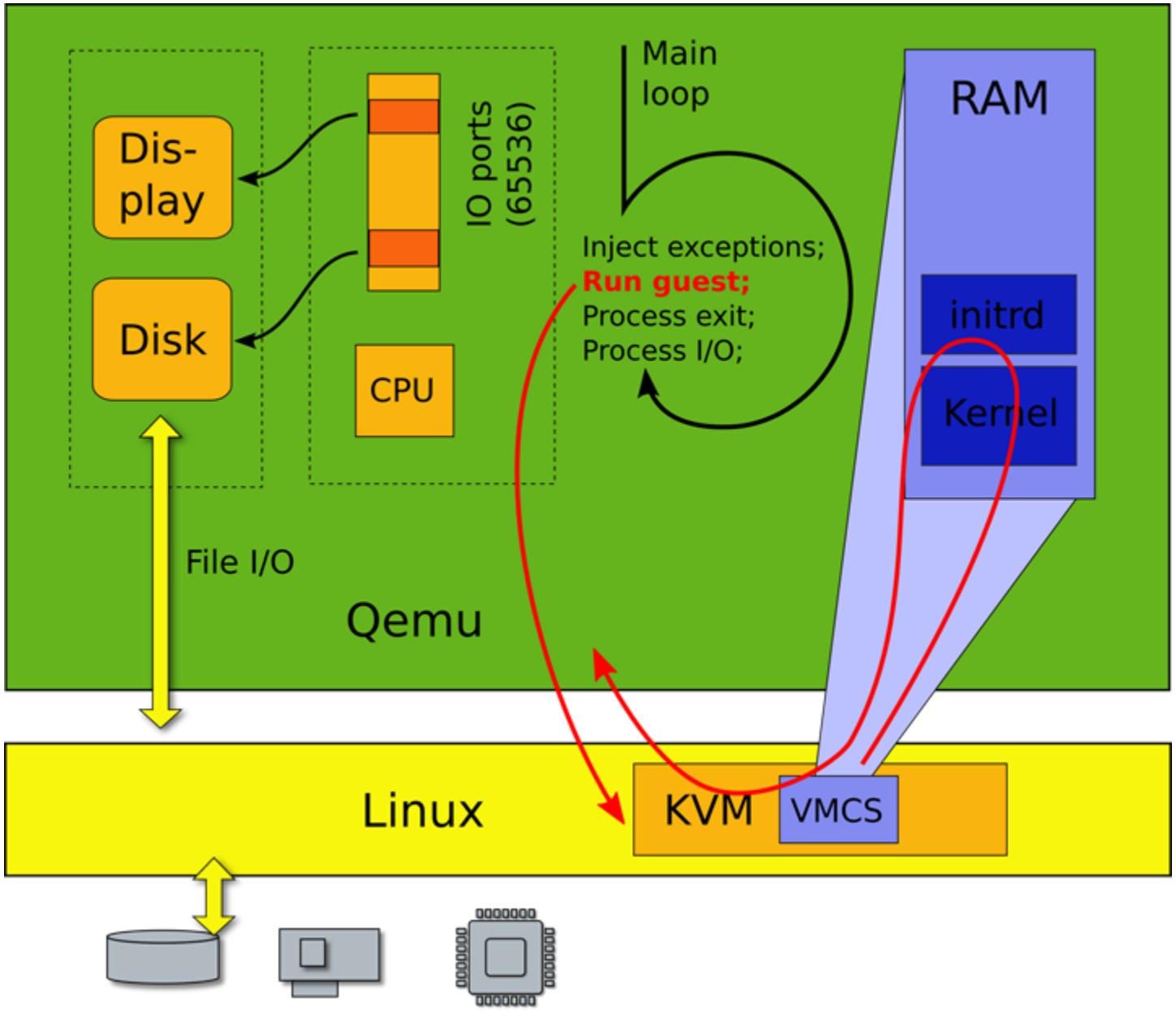
Exception bitmap

(one for each of 32 IA-32 exceptions)

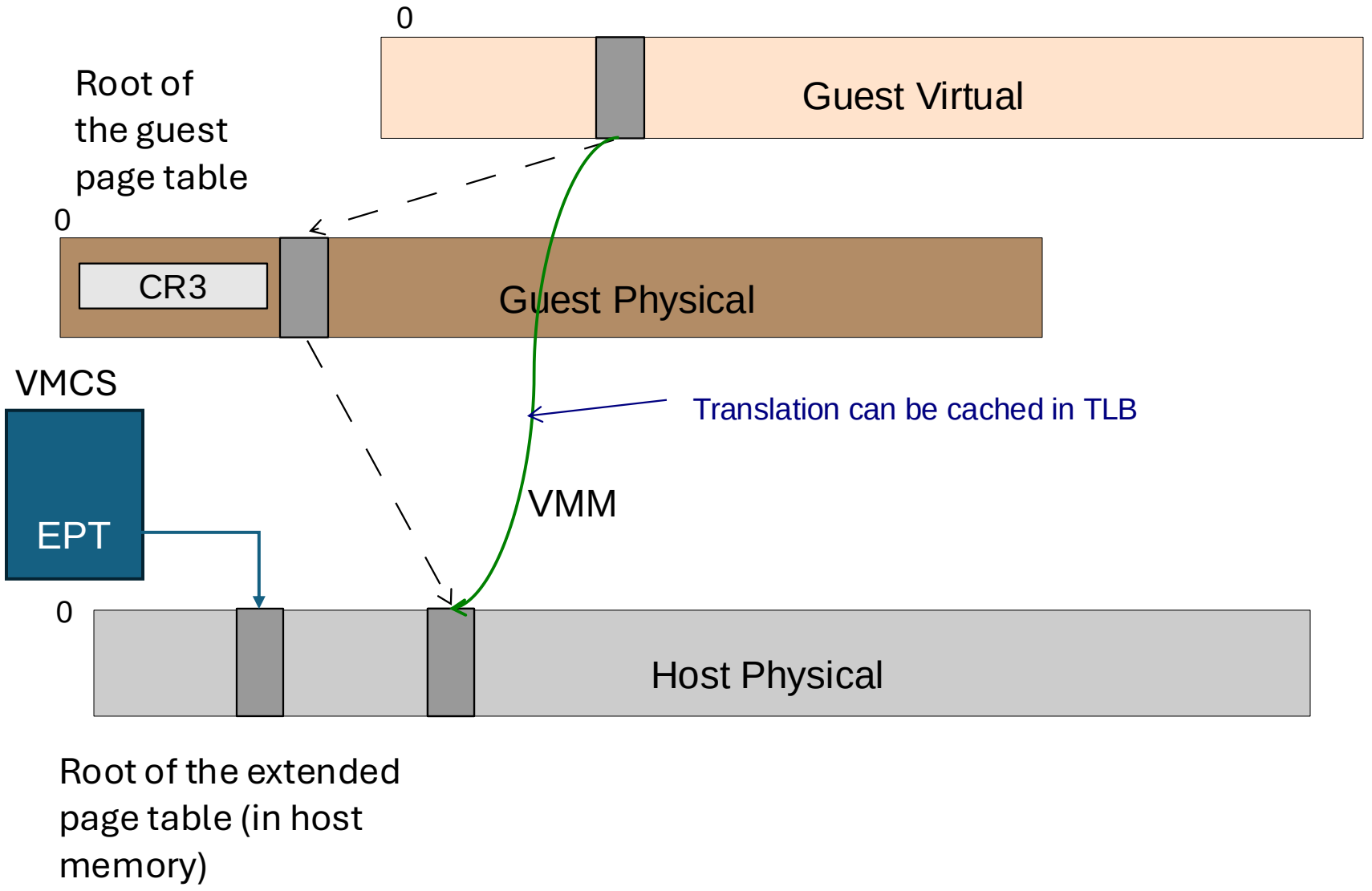
- IA-32 defines 32 exception vectors (interrupts 0-31)
- Each of them is configured to cause or not VM-exit



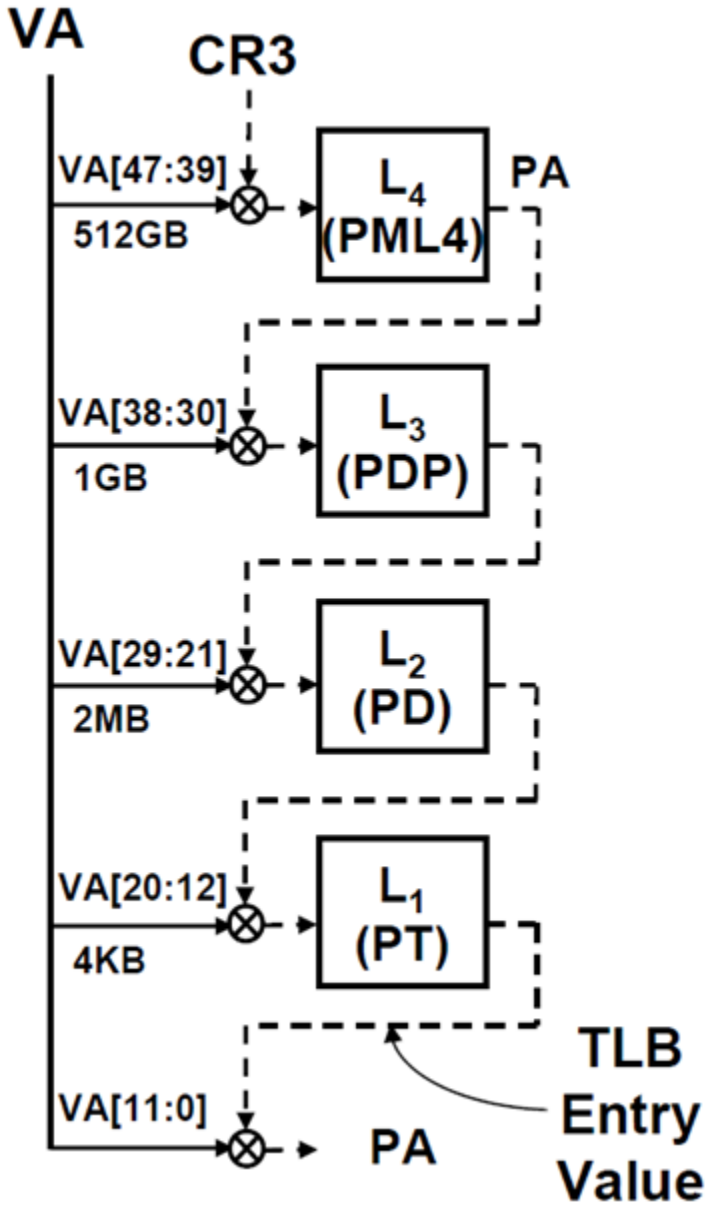
KVM with Qemu



Nested page tables

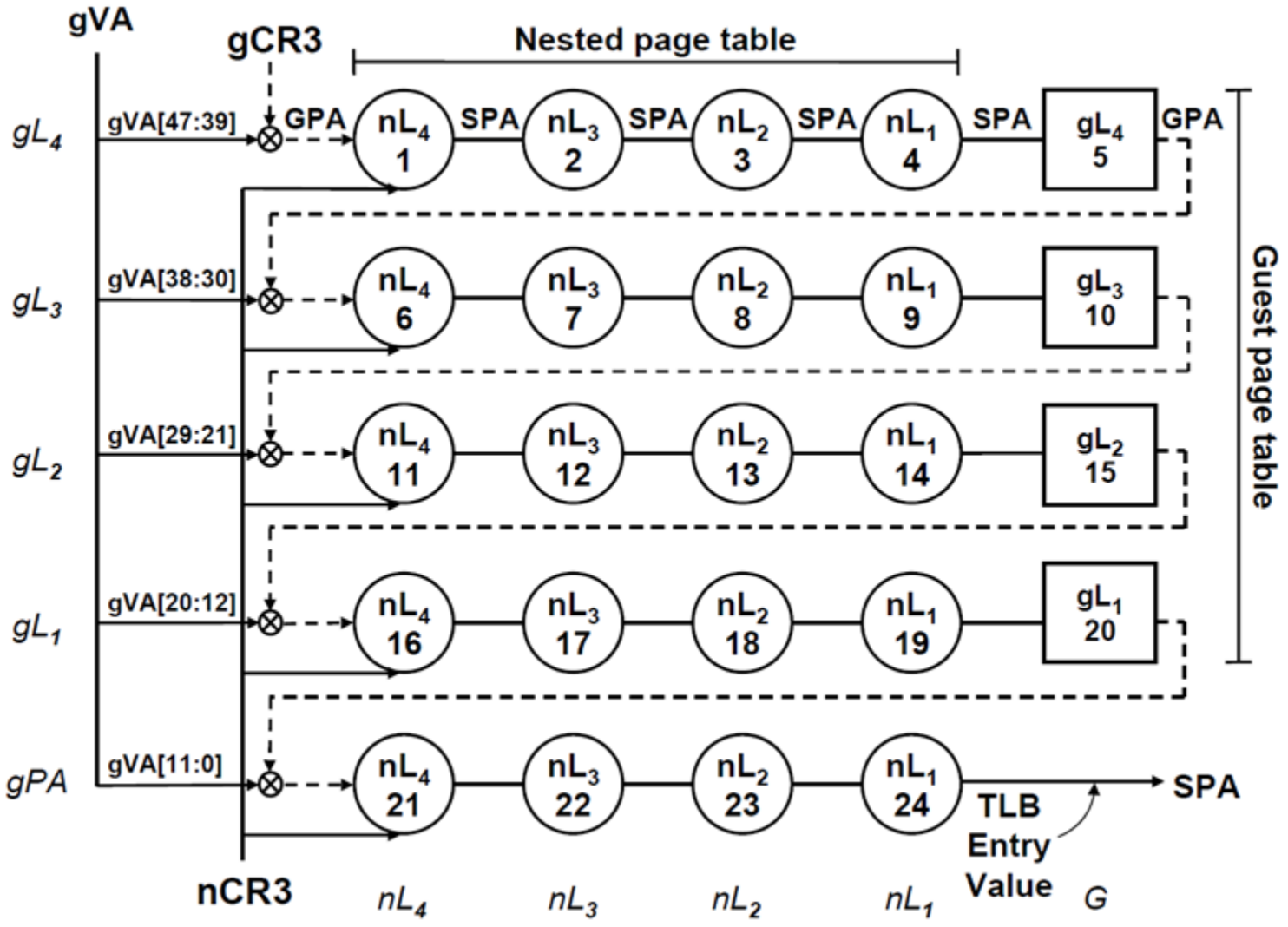


Page table lookup



- 4-level page table

Nested page table lookup



Efficient I/O

Safe Hardware Access with the Xen Virtual Machine Monitor

Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, Mark Williamson**
University of Cambridge Computer Laboratory, J J Thomson Avenue, Cambridge, UK
{firstname.lastname}@cl.cam.ac.uk

Abstract

The Xen virtual machine monitor allows multiple operating systems to execute concurrently on commodity x86 hardware, providing a solution for server consolidation and utility computing. In our initial design, Xen itself contained device-driver code and provided safe shared virtual device access. In this paper we present our new *Safe Hardware Interface*, an isolation architecture used within the latest release of Xen which allows unmodified device drivers to be shared across isolated operating system instances, while protecting individual OSs, and the system as a whole, from driver failure.

1 Introduction

We have recently developed Xen [1], an x86-based virtual machine manager specifically targeting two utility-based computing environments:

1. organizational compute/data centers, where a large cluster of physical machines may be shared across different administrative units; and

To achieve this robustly, we have developed a *Safe Hardware Interface* which allows the containment of practically all driver failures by limiting the driver's access to the specific hardware resources (memory, interrupts, and I/O ports) necessary for its operation. Our model, which places device drivers in separate virtual OS instances, provides two principal benefits: First, drivers are *isolated* from the rest of the system; they may crash or be intentionally restarted with minimal impact on running OSes. Second, a *unified interface* to each driver means that drivers may be safely shared across many OSes at once, and that only a single low-level driver implementation is required to support different paravirtualized operating systems.

While general approaches to partitioning and protecting systems for reliability have been explored over the past thirty years, they often depend on specific hardware support [3], or are unconcerned with enterprise-class performance and dependability [4, 5]. Our work addresses the problem of ensuring the reliability of shared device drivers for enterprise services on the PC architecture without requiring specialized hardware support.

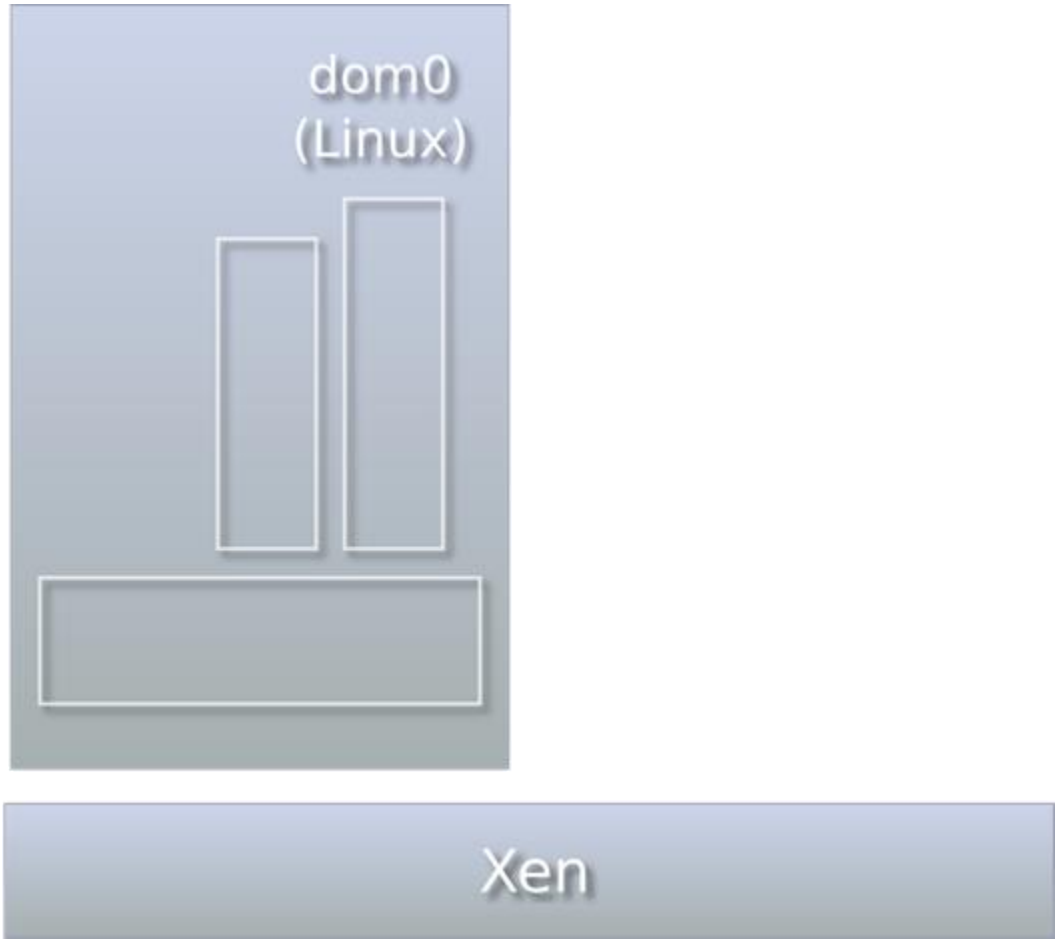
https://www.cl.cam.ac.uk/research/srg/netos/papers/2004-safehw-oasis.pdf

Where is the bottleneck

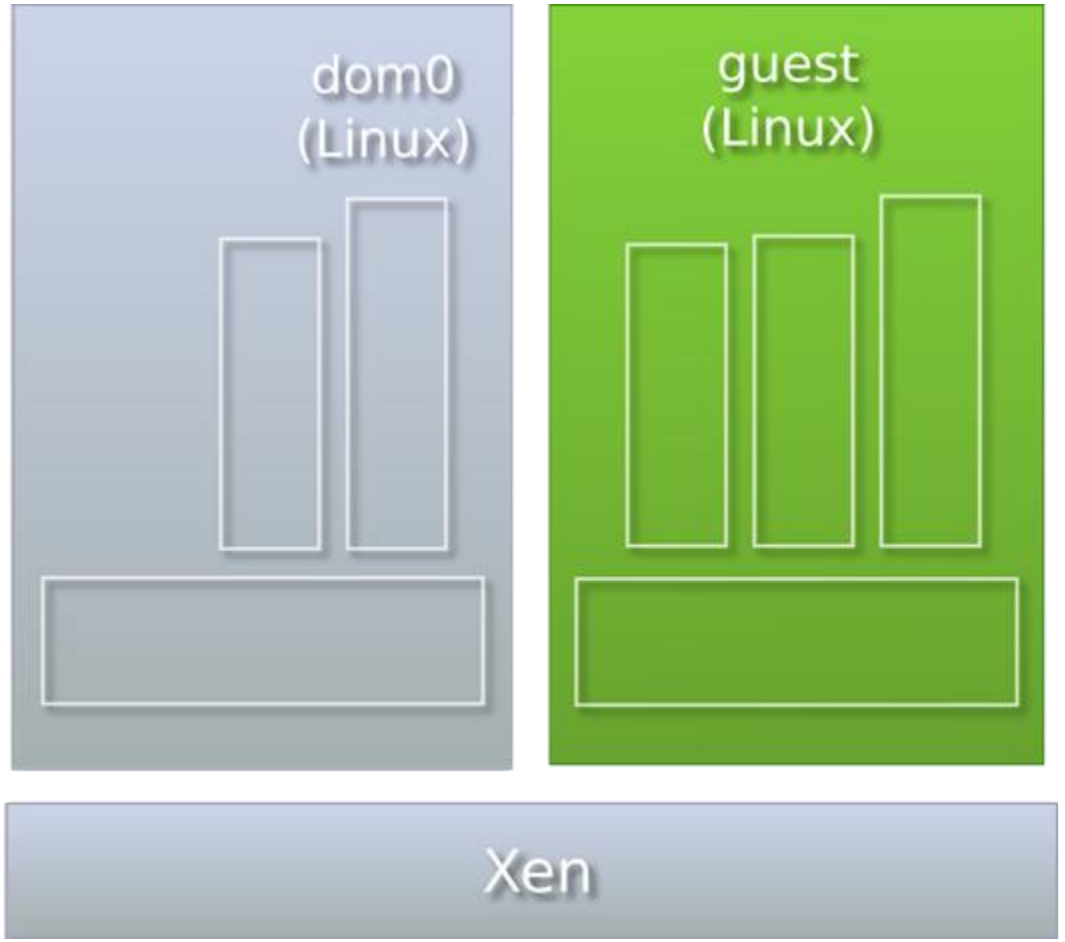
- What is the bottleneck in case of virtualization?
- CPU?
  - CPU bound workloads execute natively on the real CPU
  - Sometimes JIT compilation (binary translation makes them even faster [Dynamo]
- Everything what is inside VM is fast!
- What is the most frequent operation disturbing execution of VM?
- Device I/O!
  - Disk, Network, Graphics

Virtual devices in Xen

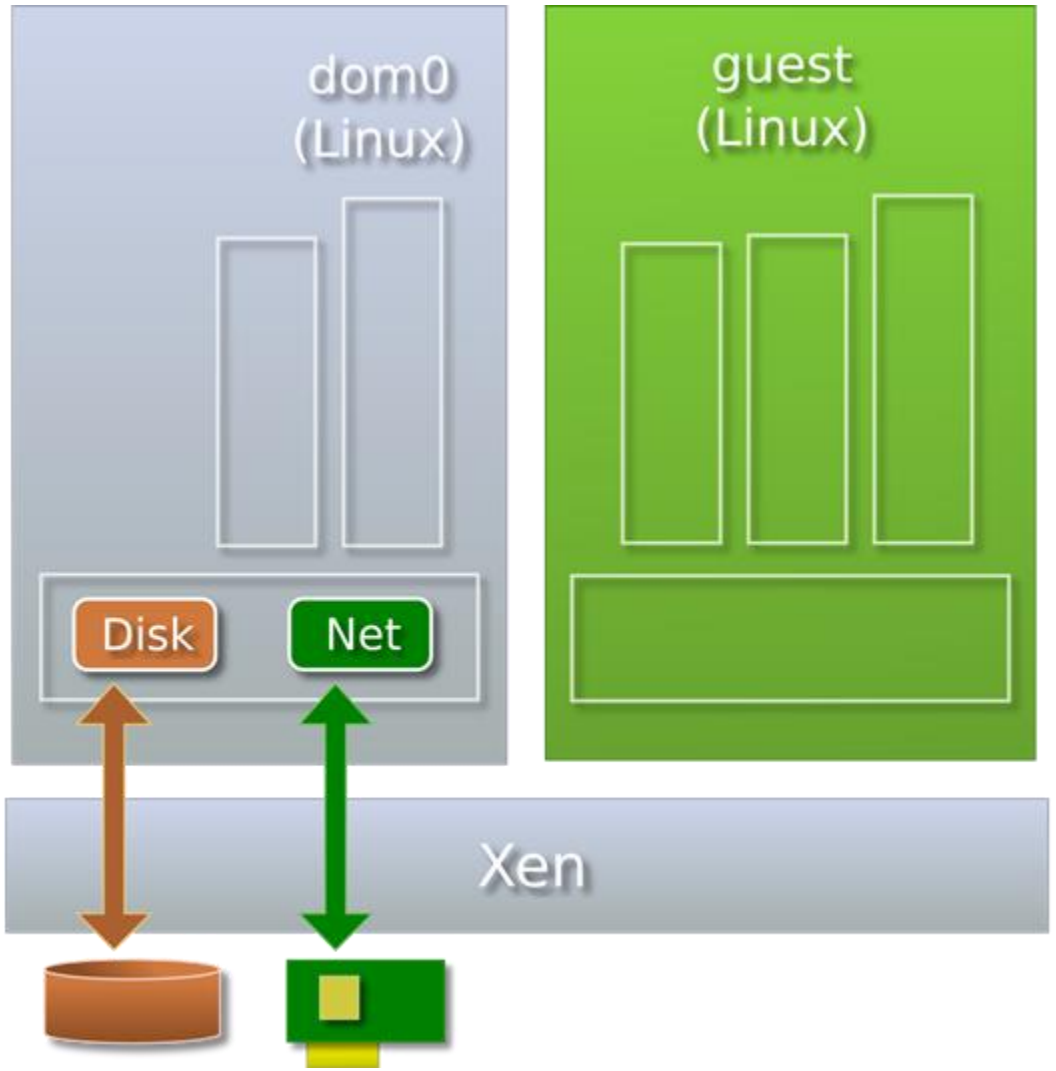
Virtual devices in Xen



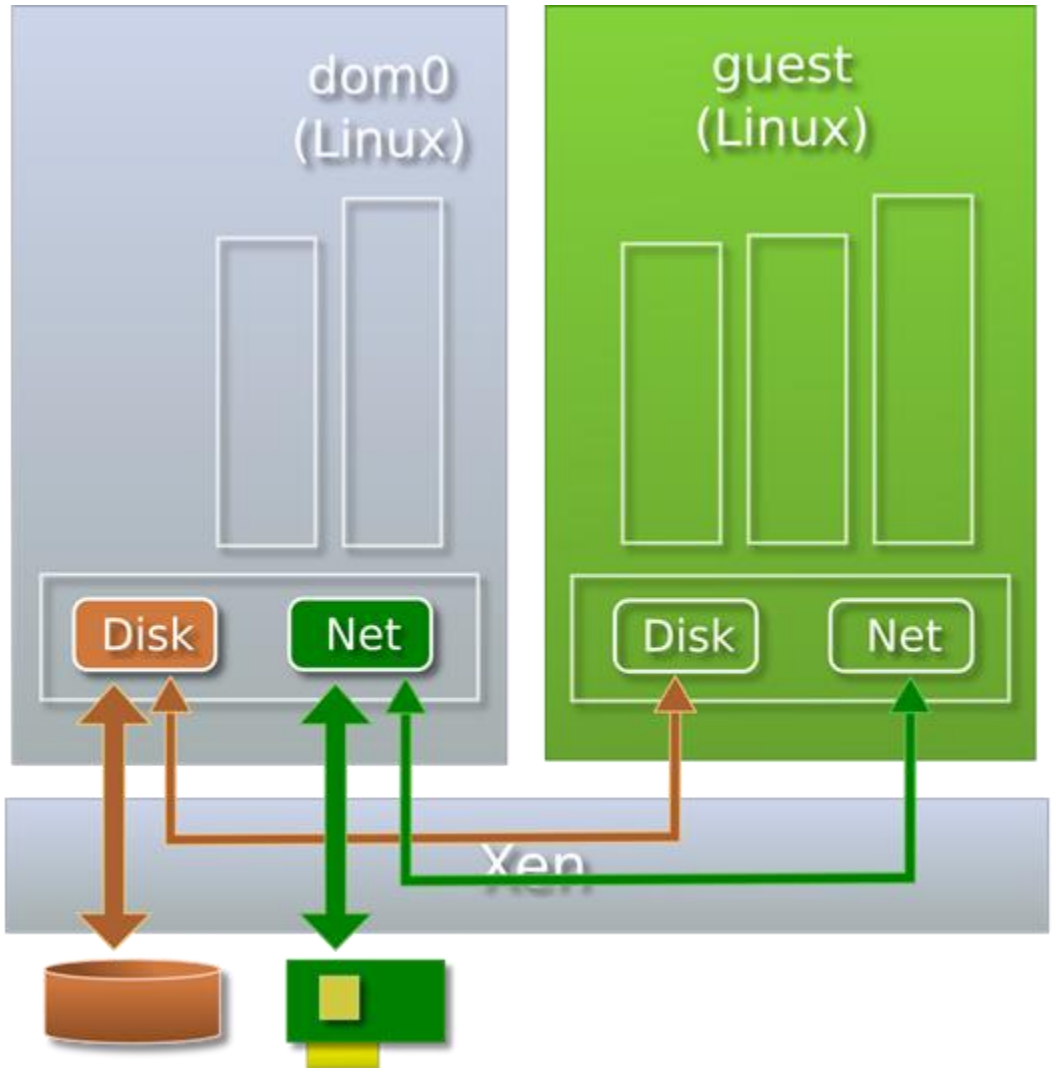
Virtual devices in Xen



Virtual devices in Xen



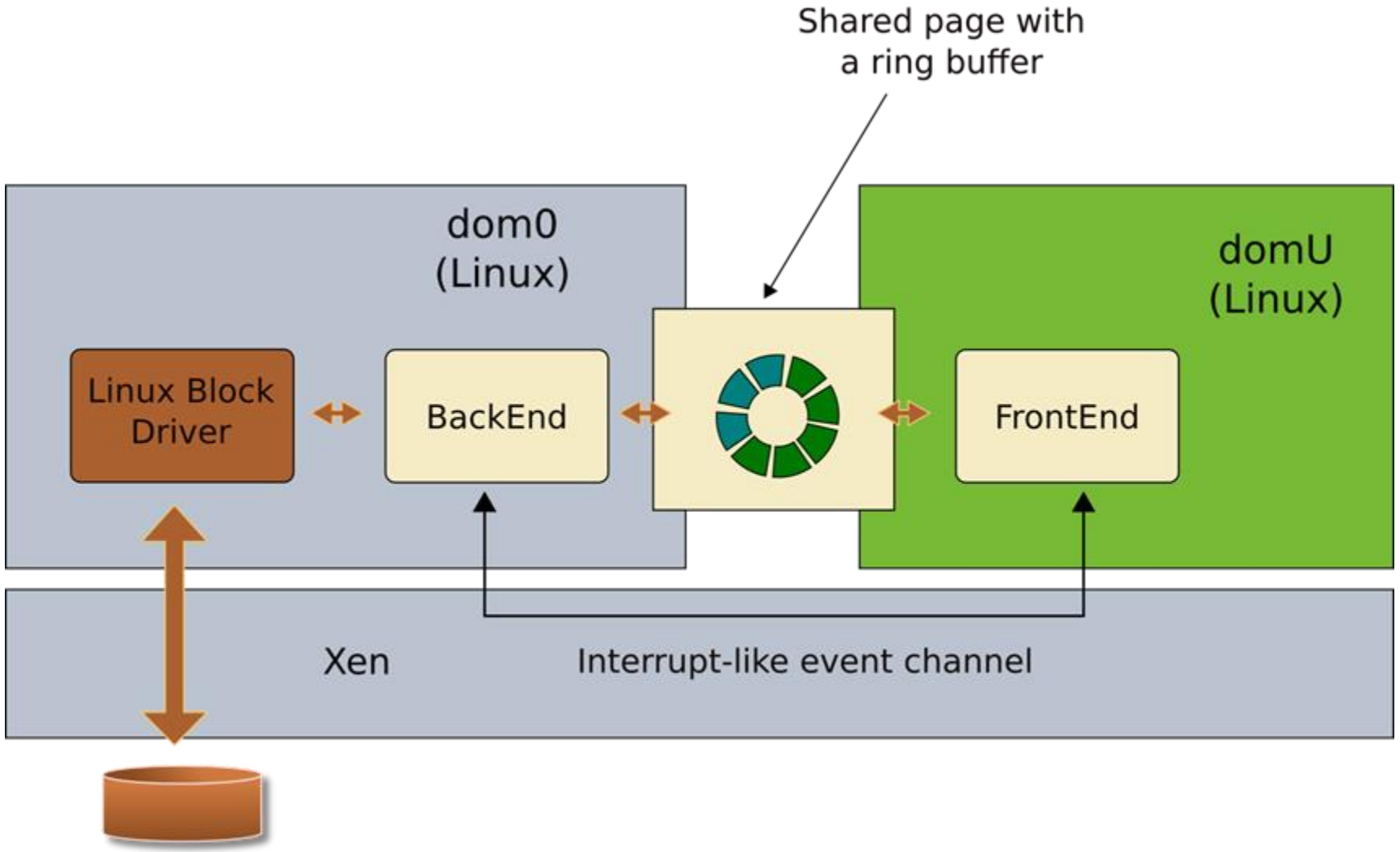
Virtual devices in Xen



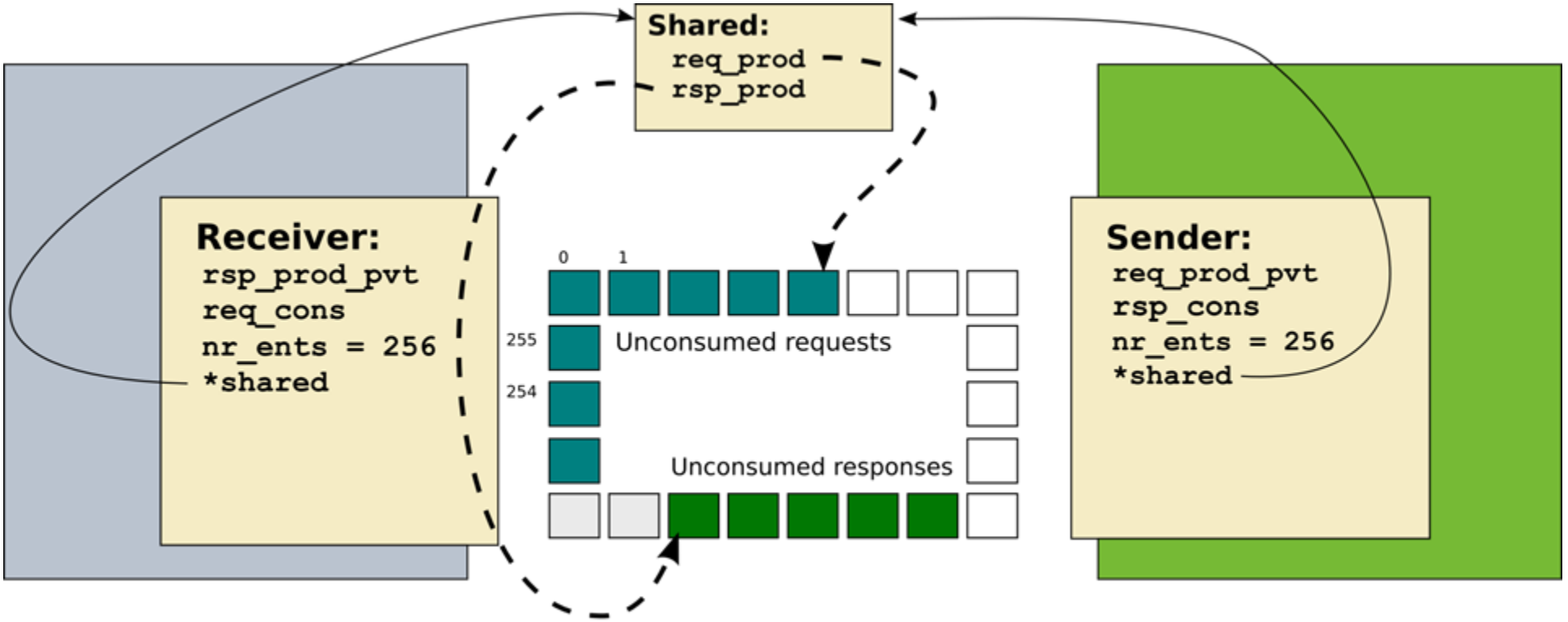
How to make the I/O fast?

- Take into account specifics of the device-driver communication
- Bulk
  - Large packets (512B – 4K)
- Session oriented
  - Connection is established once (during boot)
  - No short IPCs, like function calls
  - Costs of establishing an IPC channel are irrelevant
- Throughput oriented
  - Devices have high delays anyway
- Asynchronous
  - Again, no function calls, devices are already asynchronous

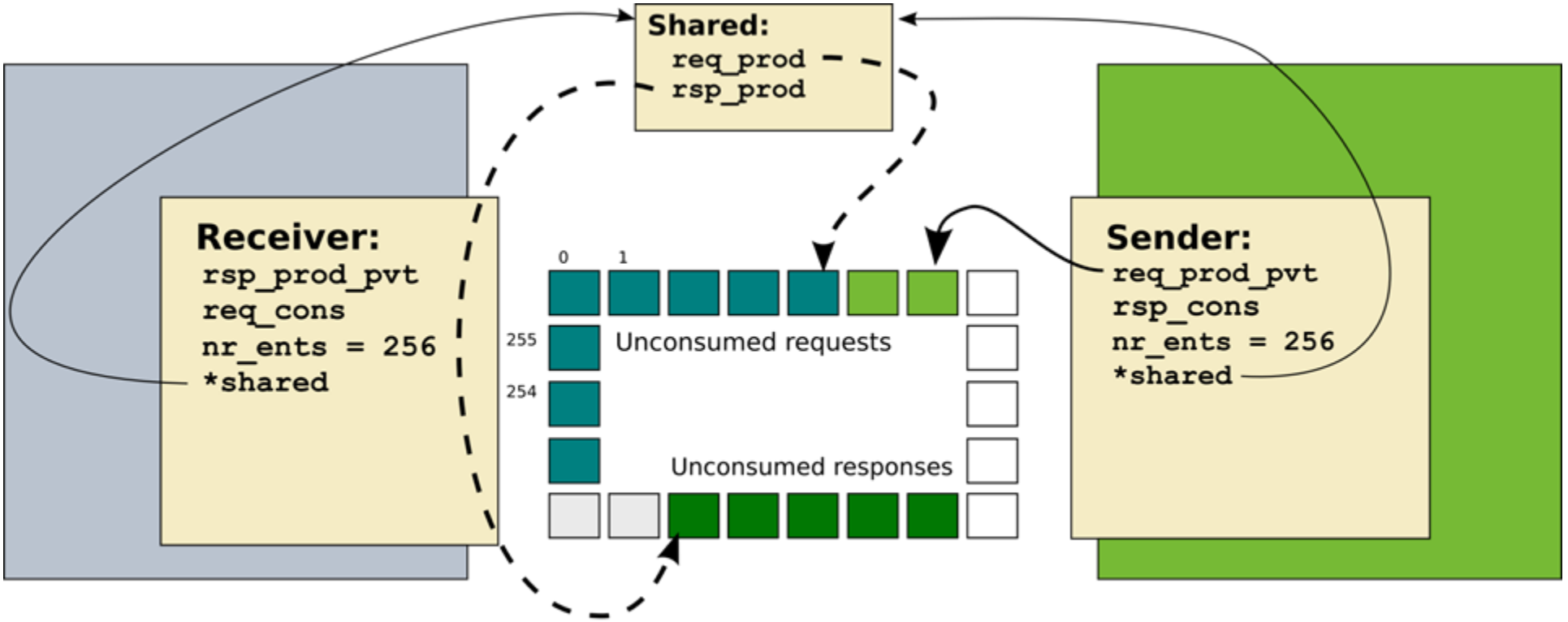
Shared rings and events



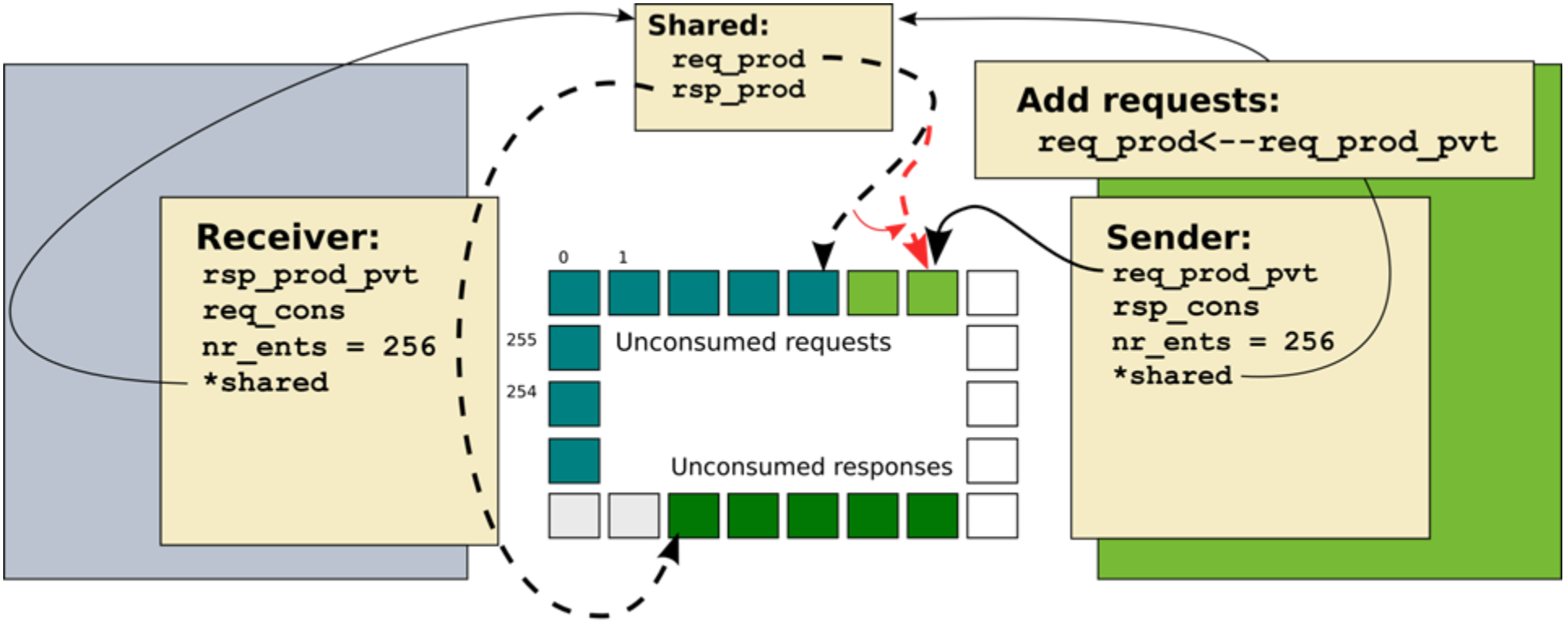
Shared rings



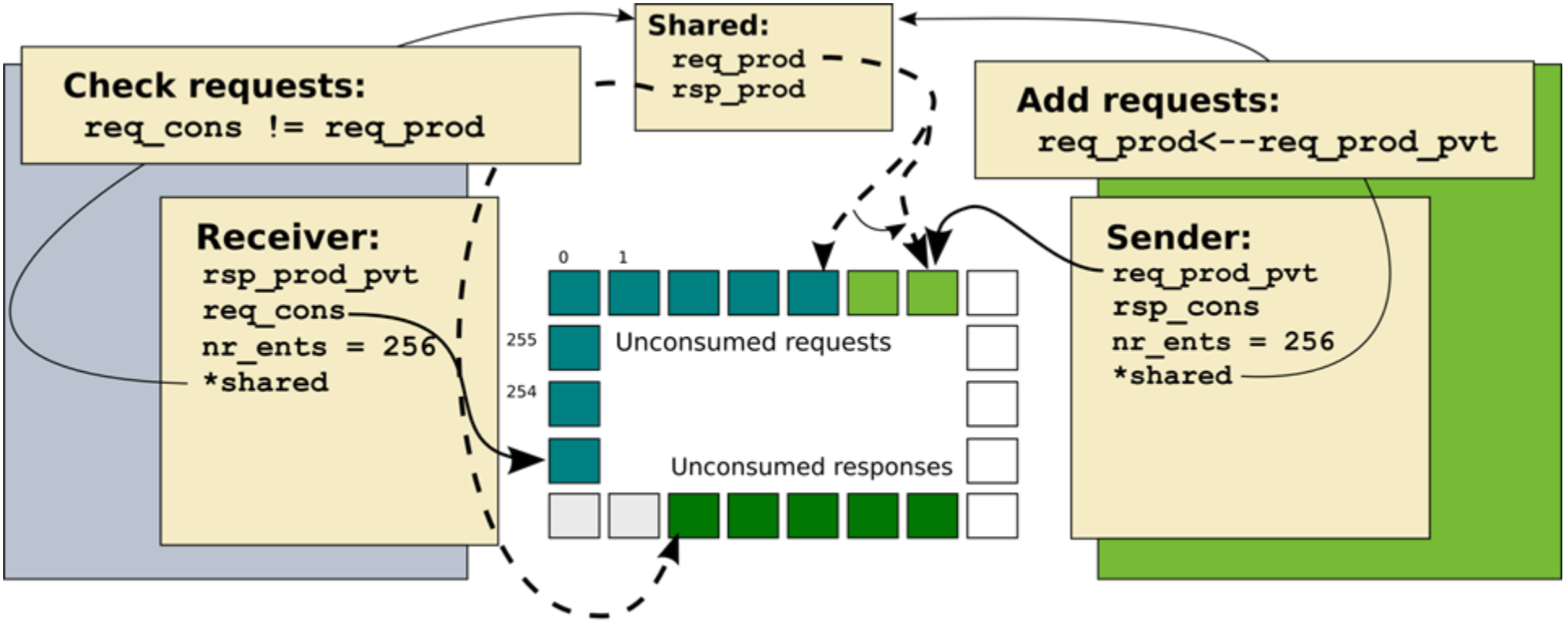
Shared rings



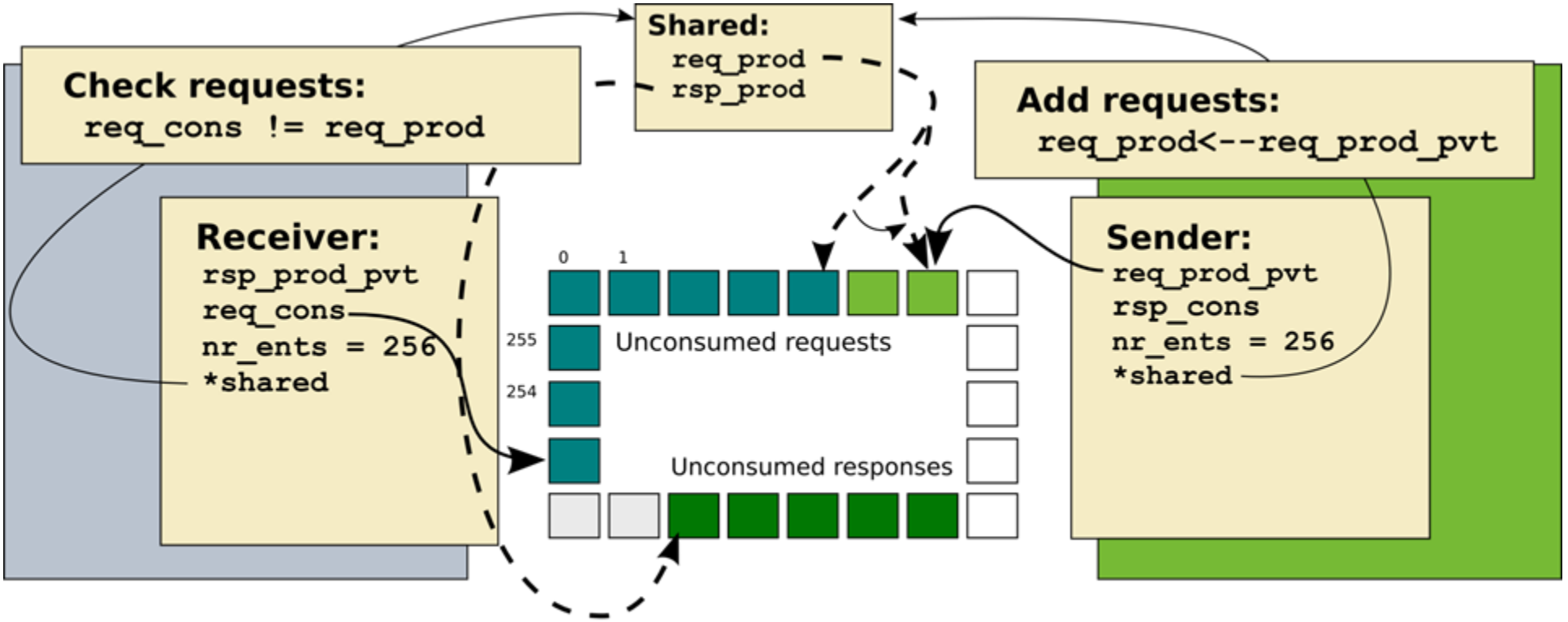
Shared rings



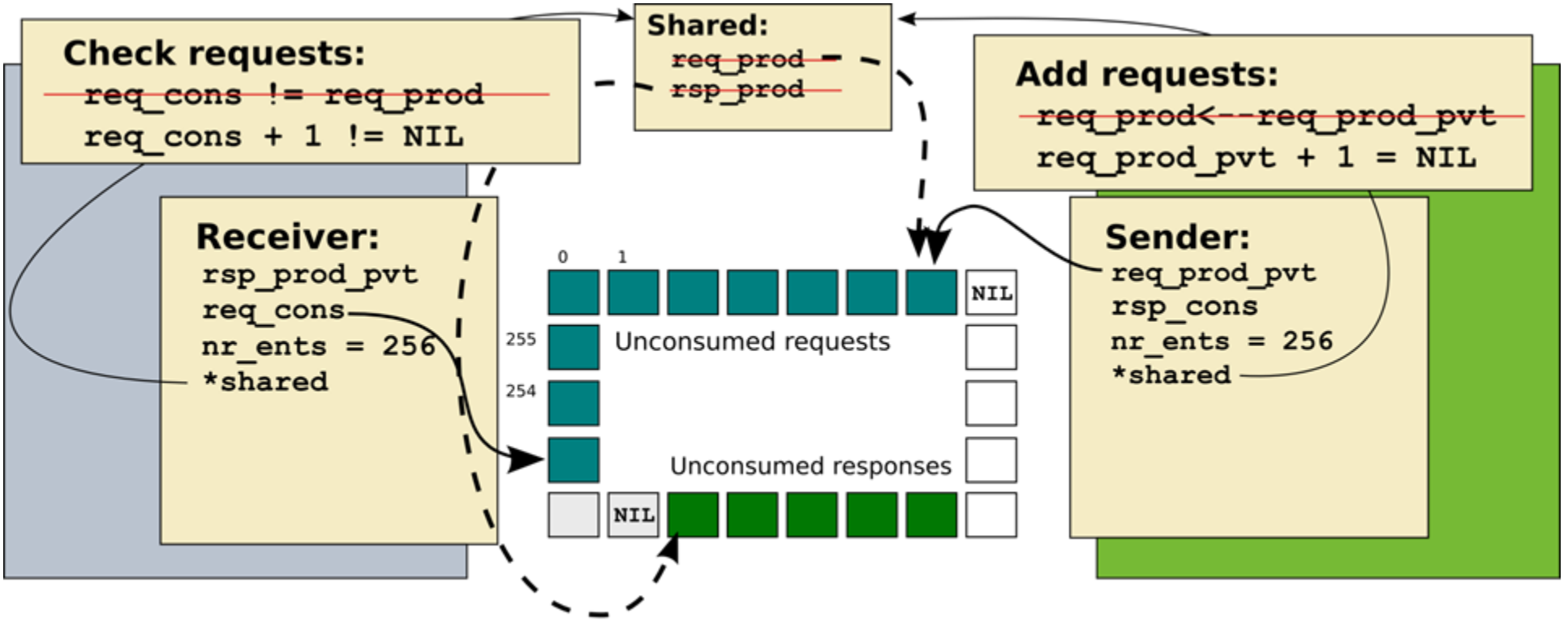
Shared rings



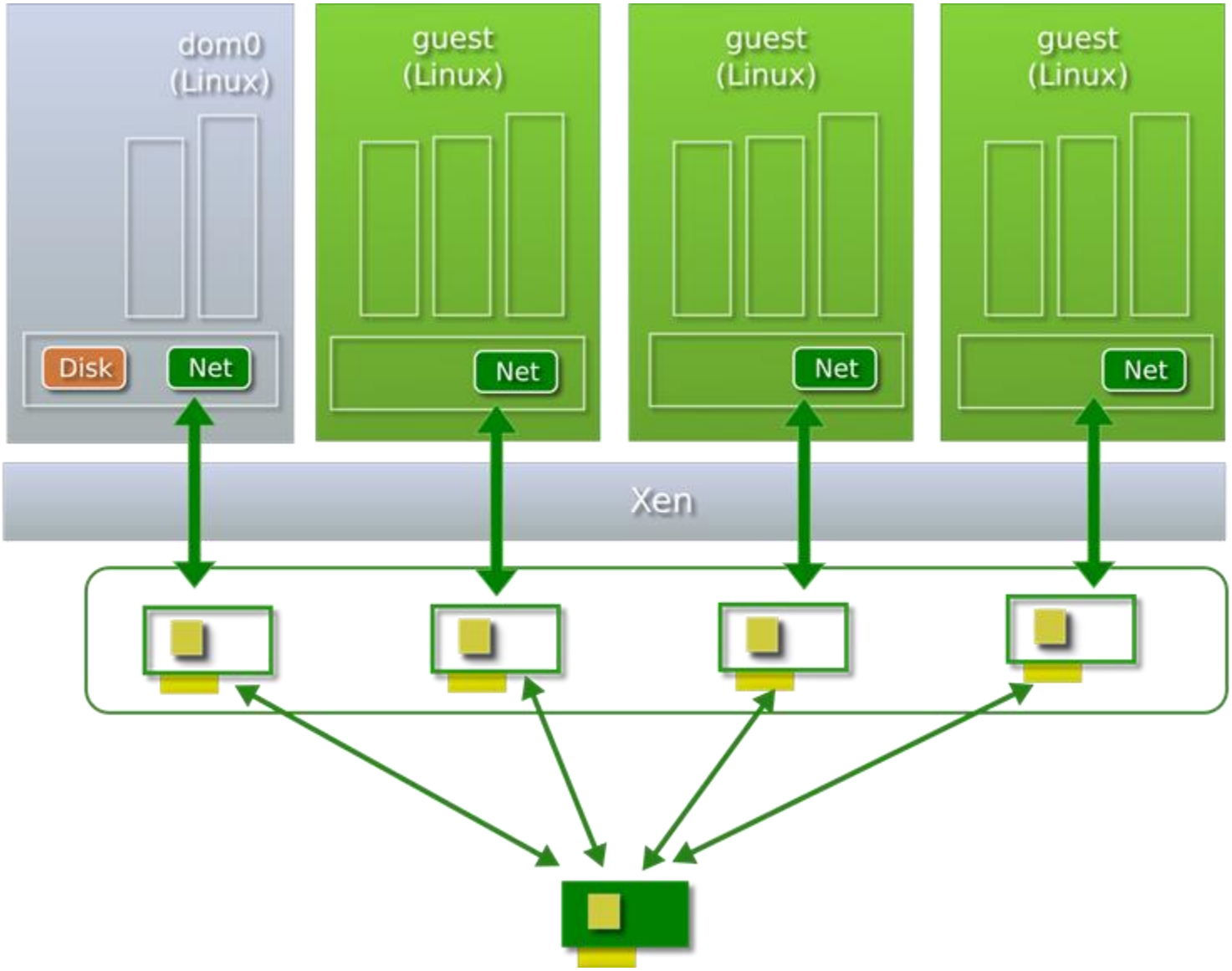
Where is a performance bottleneck here?



Eliminate cache thrashing

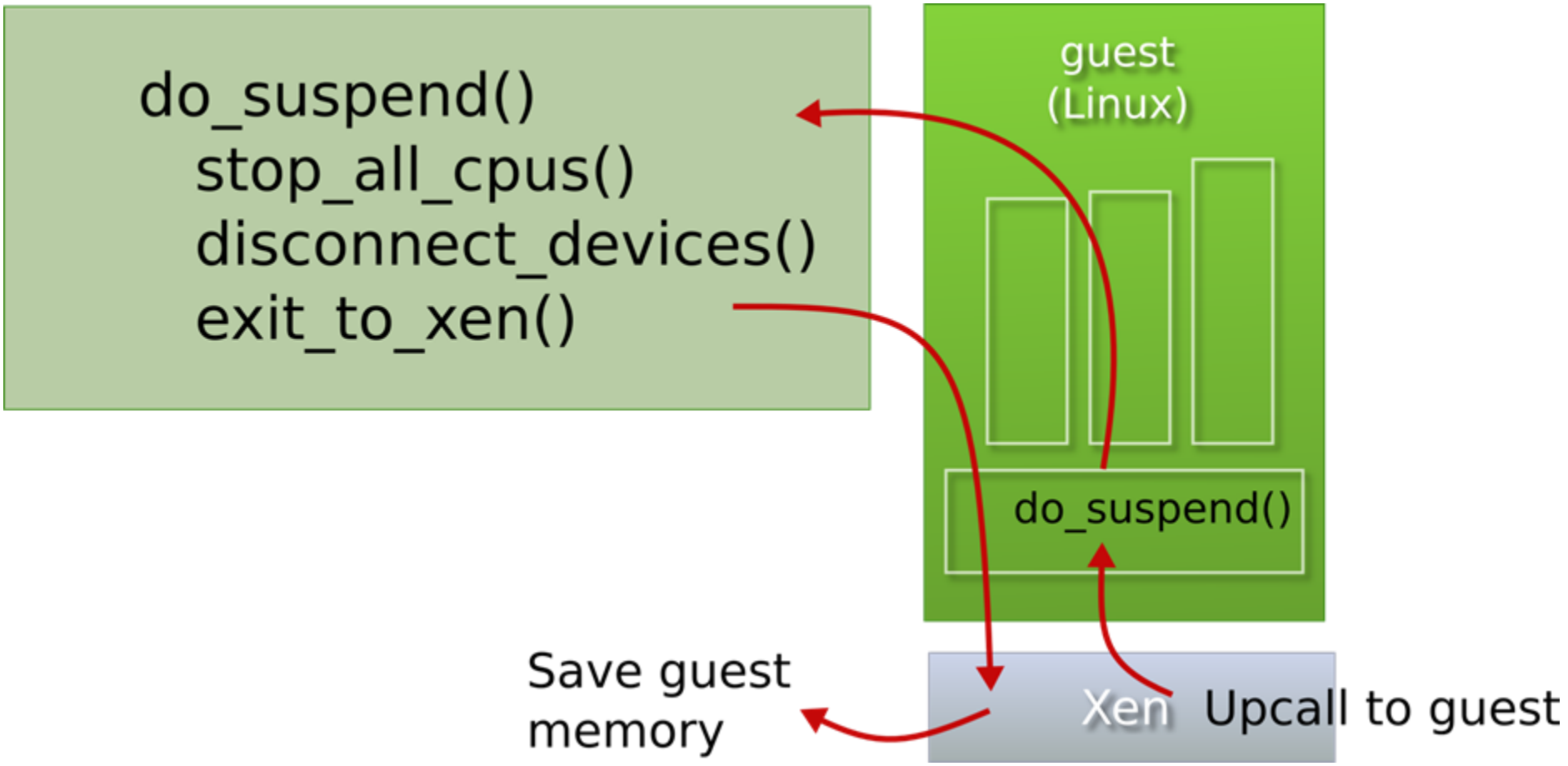


Devices supporting virtualization

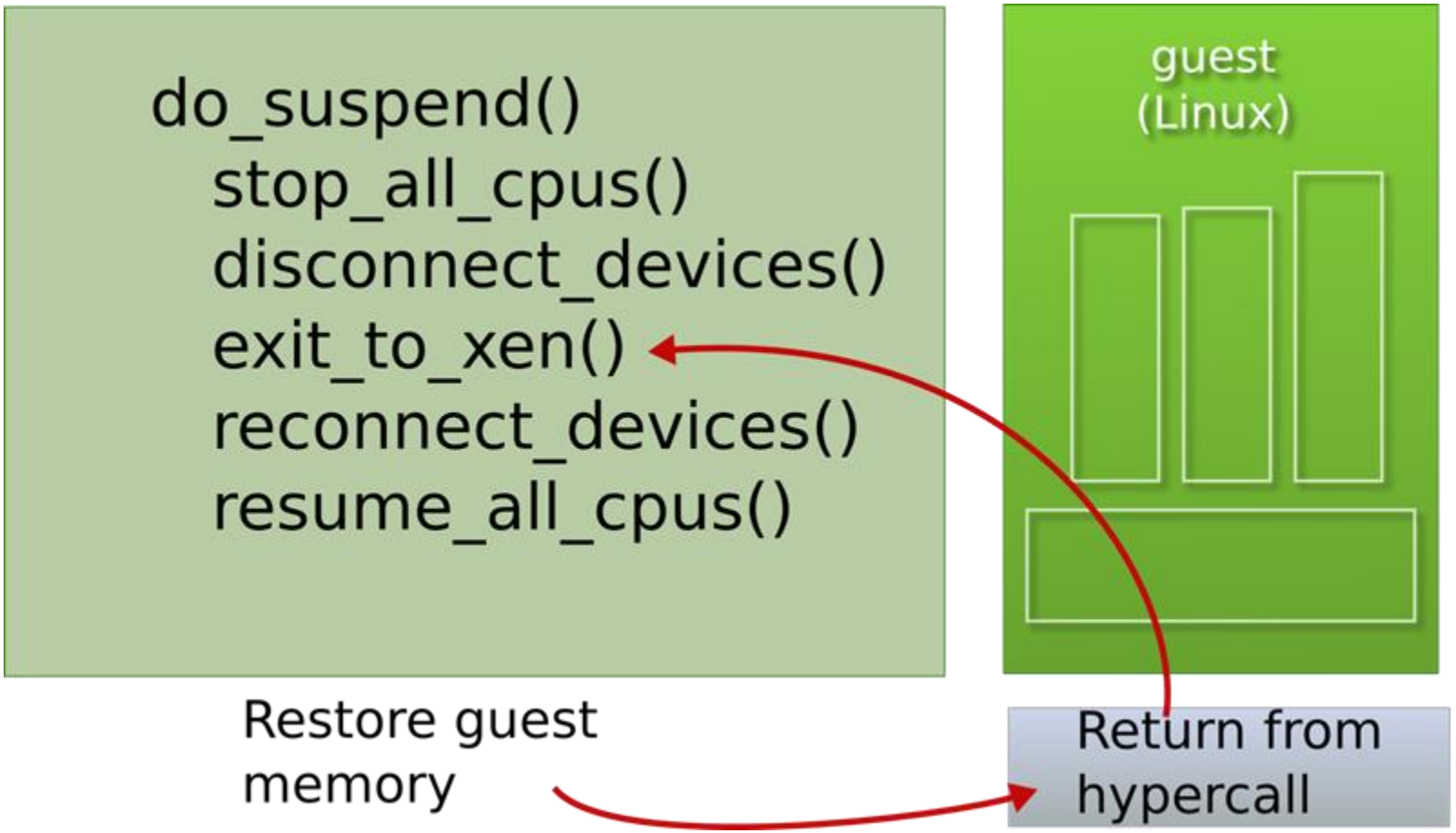


Some VM tricks: suspend/resume, checkpoints migration

Suspend


do_suspend()
stop_all_cpus()
disconnect_devices()
exit_to_xen()
guest
(Linux)
do_suspend()
Save guest
memory
Xen
Upcall to guest

Resume


do_suspend()
stop_all_cpus()
disconnect_devices()
exit_to_xen()
reconnect_devices()
resume_all_cpus()
guest
(Linux)
Restore guest
memory
Return from
hypercall

Checkpoints

- Checkpoints are almost suspend/resume
- A copy of the entire VM's state has to be saved
- Memory
  - OK, it's relatively small 128MB-4GB
- Disk
  - Problem: disks are huge 100GB-1TB

- How to save storage efficiently?

Parallax: Virtual Disks for Virtual Machines

Dutch T. Meyer, Gitika Aggarwal, Brendan Cully, Geoffrey Lefebvre,
Michael J. Feeley, Norman C. Hutchinson, and Andrew Warfield*

{dmeyer, gitika, brendan, geoffrey, feeley, norm, andy}@cs.ubc.ca
Department of Computer Science
University of British Columbia
Vancouver, BC, Canada

ABSTRACT

Parallax is a distributed storage system that uses virtualization to provide storage facilities specifically for virtual environments. The system employs a novel architecture in which storage features that have traditionally been implemented directly on high-end storage arrays and switches are relocated into a federation of *storage VMs*, sharing the same physical hosts as the VMs that they serve. This architecture retains the single administrative domain and OS agnosticism achieved by array- and switch-based approaches, while lowering the bar on hardware requirements and facilitating the development of new features. Parallax offers a comprehensive set of storage features including frequent, low-overhead snapshot of virtual disks, the "gold-mastering" of template images, and the ability to use local disks as a persistent cache to dampen burst demand on networked storage.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management—*Storage Hierarchies*; D.4.7 [**Operating Systems**]: Organization and Design—*Distributed Systems*

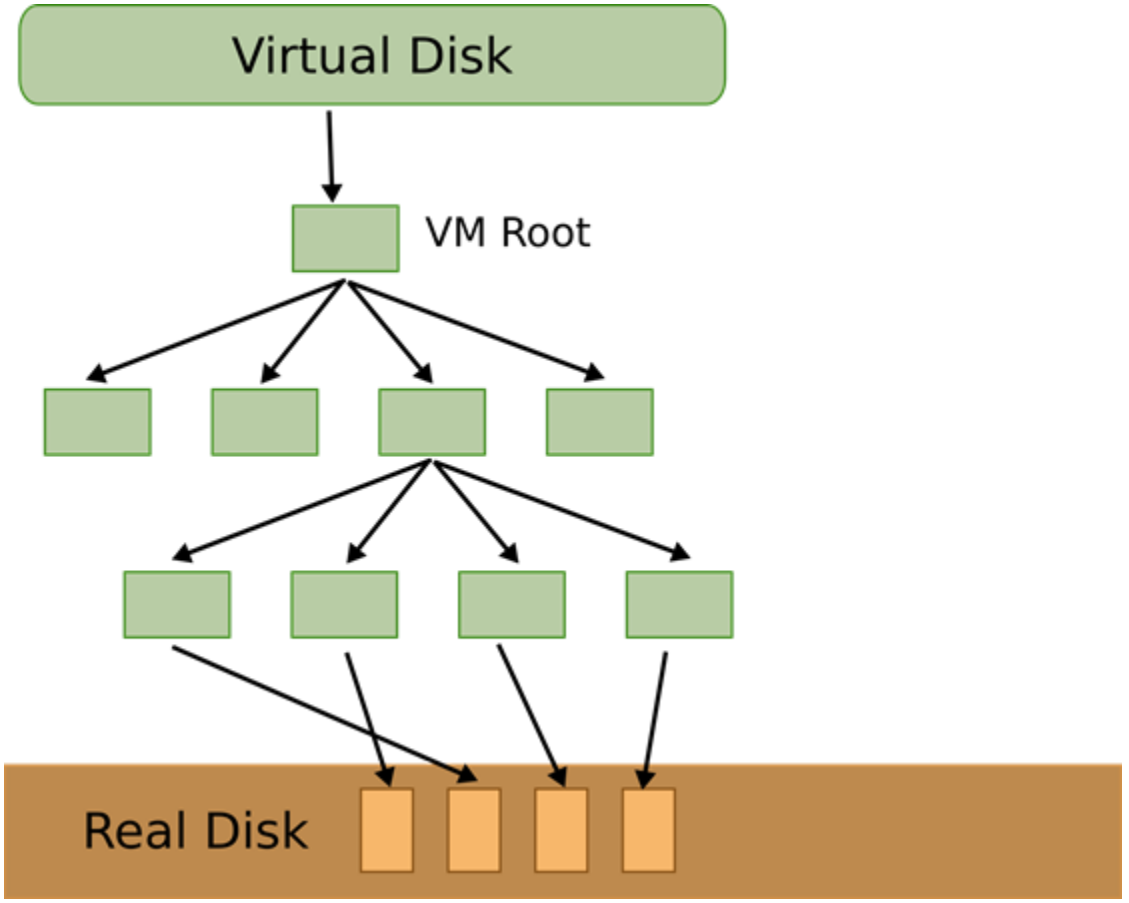
this encapsulation have been demonstrated by numerous interesting research projects that allow VMs to travel through space [24, 2, 13], time [4, 12, 32], and to be otherwise manipulated [30].

Unfortunately, while both system software and platform hardware such as CPUs and chipsets have evolved rapidly in support of virtualization, storage has not. While "storage virtualization" is widely available, the term is something of a misnomer in that it is largely used to describe the aggregation and repartitioning of disks at very coarse time scales for use by physical machines. VM deployments are limited by modern storage systems because the storage primitives available for use by VMs are not nearly as nimble as the VMs that consume them. Operations such as remapping volumes across hosts and checkpointing disks are frequently clumsy and esoteric on high-end storage systems, and are simply unavailable on lower-end commodity storage hardware.

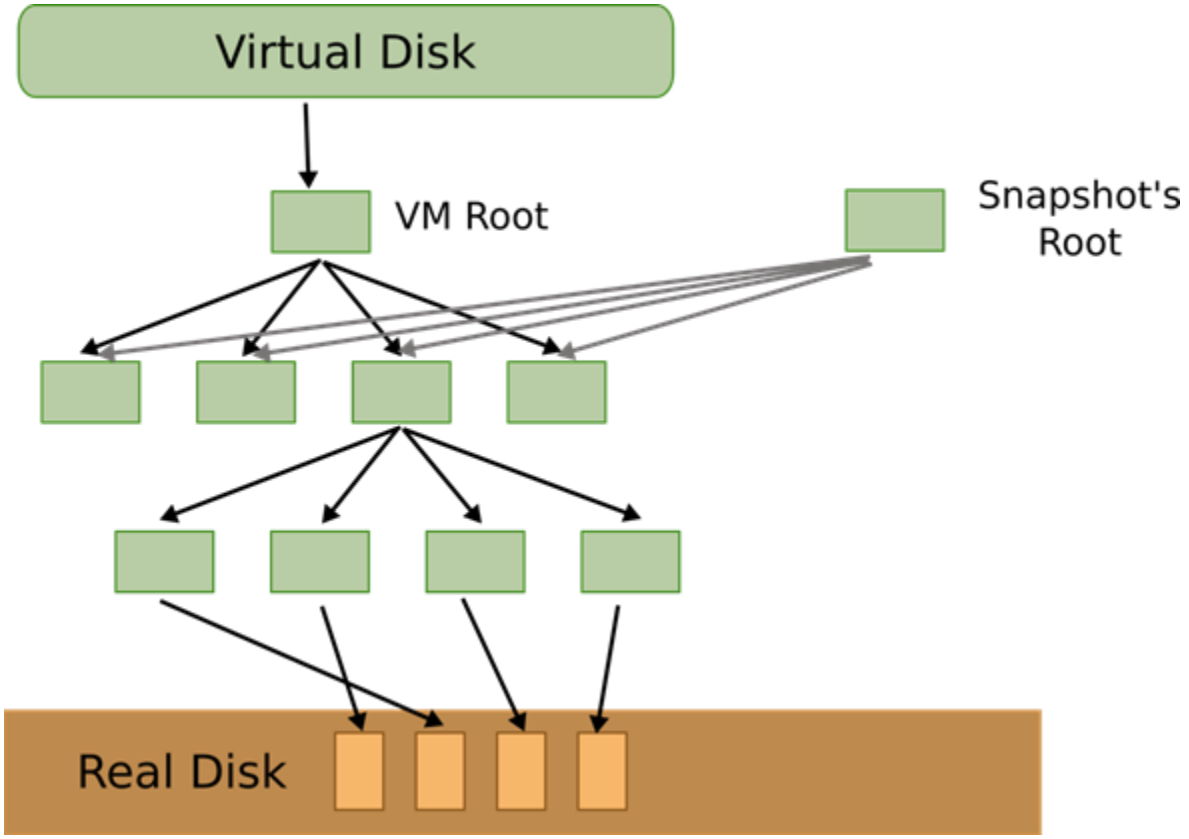
This paper describes *Parallax*, a system that attempts to *use* virtualization in order to provide advanced storage services *for* virtual machines. Parallax takes advantage of the structure of a virtualized environment to move storage enhancements that are traditionally implemented on arrays or in storage switches out onto the consuming physical hosts. Each host in a Parallax-based cluster runs a *storage VM*, which is a virtual appliance [23] specifically for stor-

https://www.cs.ubc.ca/~andy/papers/parallax-eurosys-final.pdf

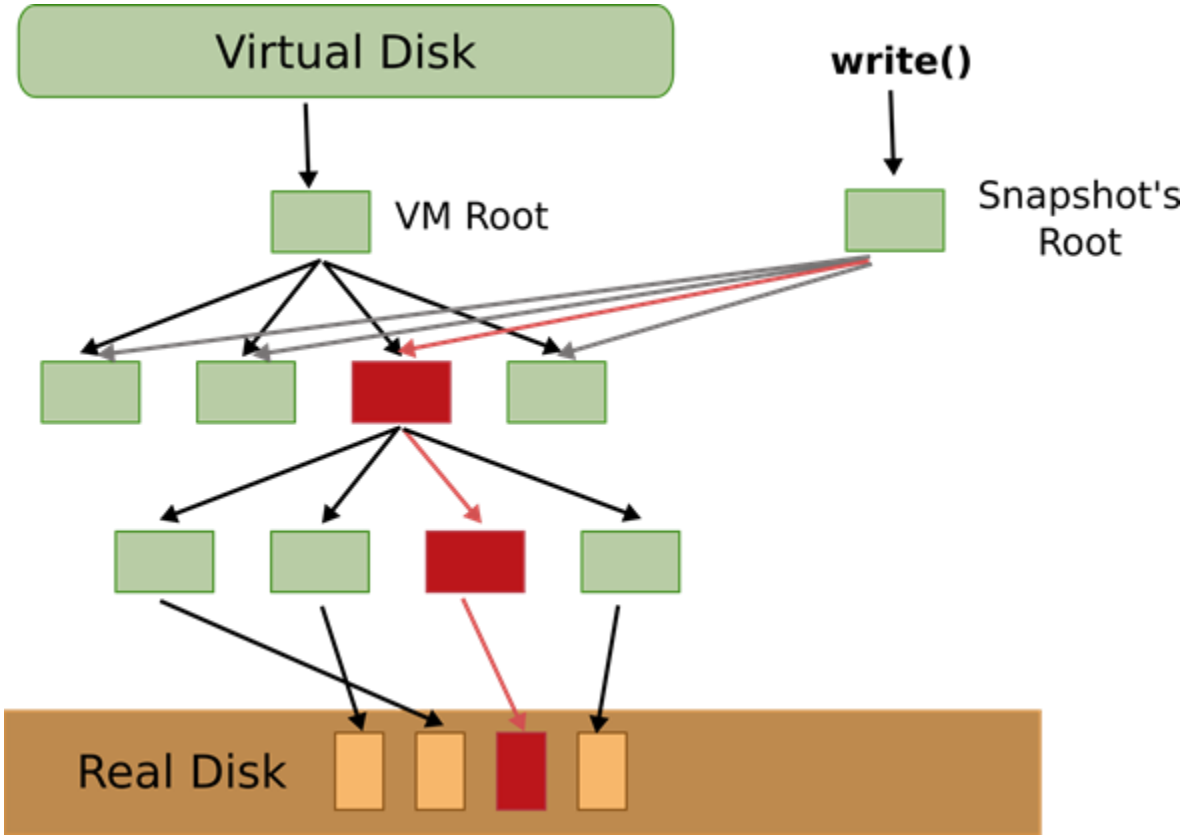
Branching storage



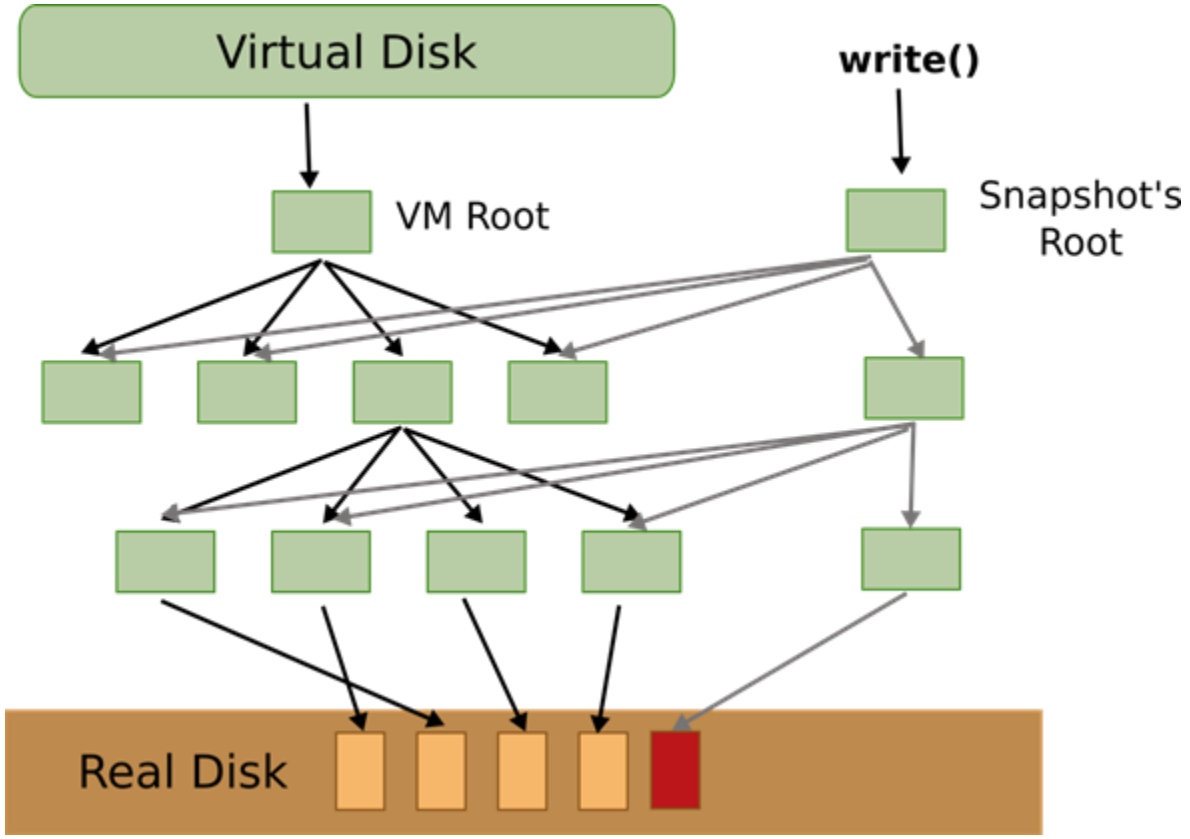
Branching storage: snapshot



Branching storage: writes



Branching storage: snapshot



Migration

- Migration is essentially a live checkpoint between machines
- The goal: minimal downtime

- How to make the checkpoint faster?

Live Migration of Virtual Machines

Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen†,
Eric Jul†, Christian Limpach, Ian Pratt, Andrew Warfield
University of Cambridge Computer Laboratory
15 JJ Thomson Avenue, Cambridge, UK
firstname.lastname@cl.cam.ac.uk
† *Department of Computer Science*
University of Copenhagen, Denmark
{jacobg,eric}@diku.dk

Abstract

Migrating operating system instances across distinct physical hosts is a useful tool for administrators of data centers and clusters: It allows a clean separation between hardware and software, and facilitates fault management, load balancing, and low-level system maintenance.

By carrying out the majority of migration while OSes continue to run, we achieve impressive performance with minimal service downtimes; we demonstrate the migration of entire OS instances on a commodity cluster, recording service downtimes as low as $60ms$. We show that that our performance is sufficient to make live migration a practical tool even for servers running interactive loads.

In this paper we consider the design options for migrating OSes running services with liveness constraints, focusing on data center and cluster environments. We introduce and analyze the concept of *writable working set*, and present the design, implementation and evaluation of high-performance OS migration built on top of the Xen VMM.

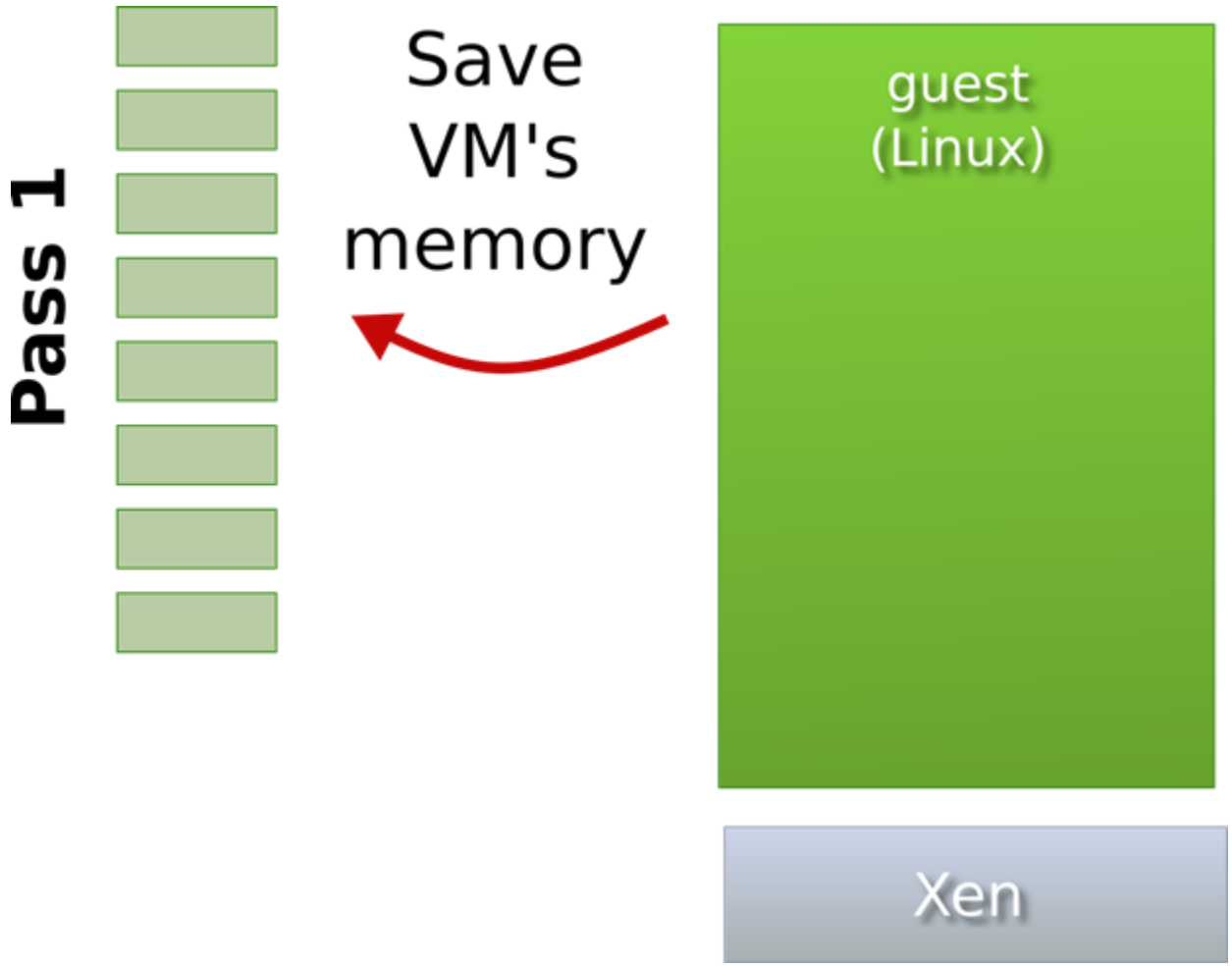
certain system calls or even memory accesses on behalf of migrated processes. With virtual machine migration, on the other hand, the original host may be decommissioned once migration has completed. This is particularly valuable when migration is occurring in order to allow maintenance of the original host.

Secondly, migrating at the level of an entire virtual machine means that in-memory state can be transferred in a consistent and (as will be shown) efficient fashion. This applies to kernel-internal state (e.g. the TCP control block for a currently active connection) as well as application-level state, even when this is shared between multiple cooperating processes. In practical terms, for example, this means that we can migrate an on-line game server or streaming media server without requiring clients to reconnect: something not possible with approaches which use application-level restart and layer 7 redirection.

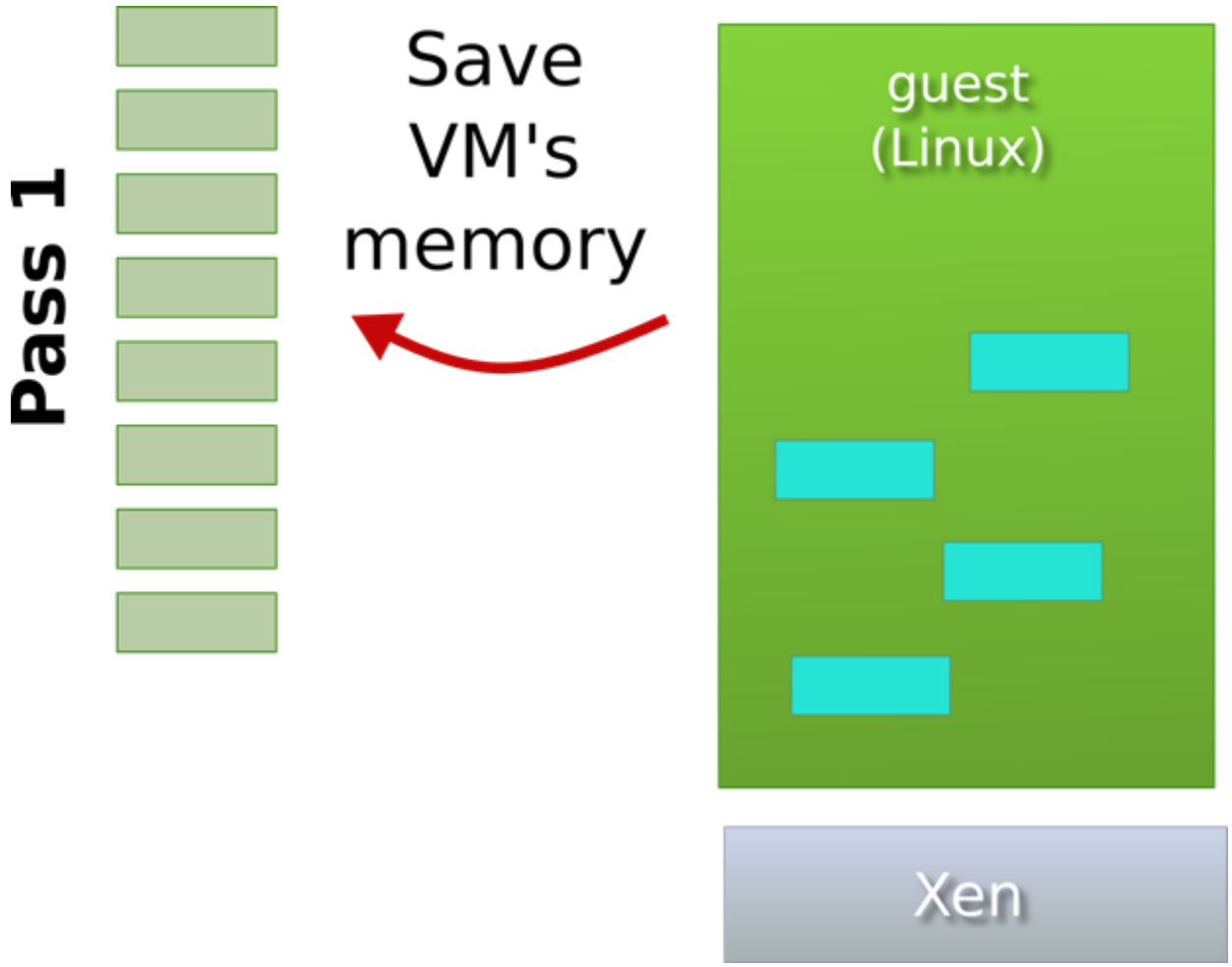
Thirdly, live migration of virtual machines allows a separation of concerns between the users and operator of a data center or cluster. Users have 'carte blanche' regarding the

https://www.usenix.org/legacy/event/nsdi05/tech/full_papers/clark/clark.pdf

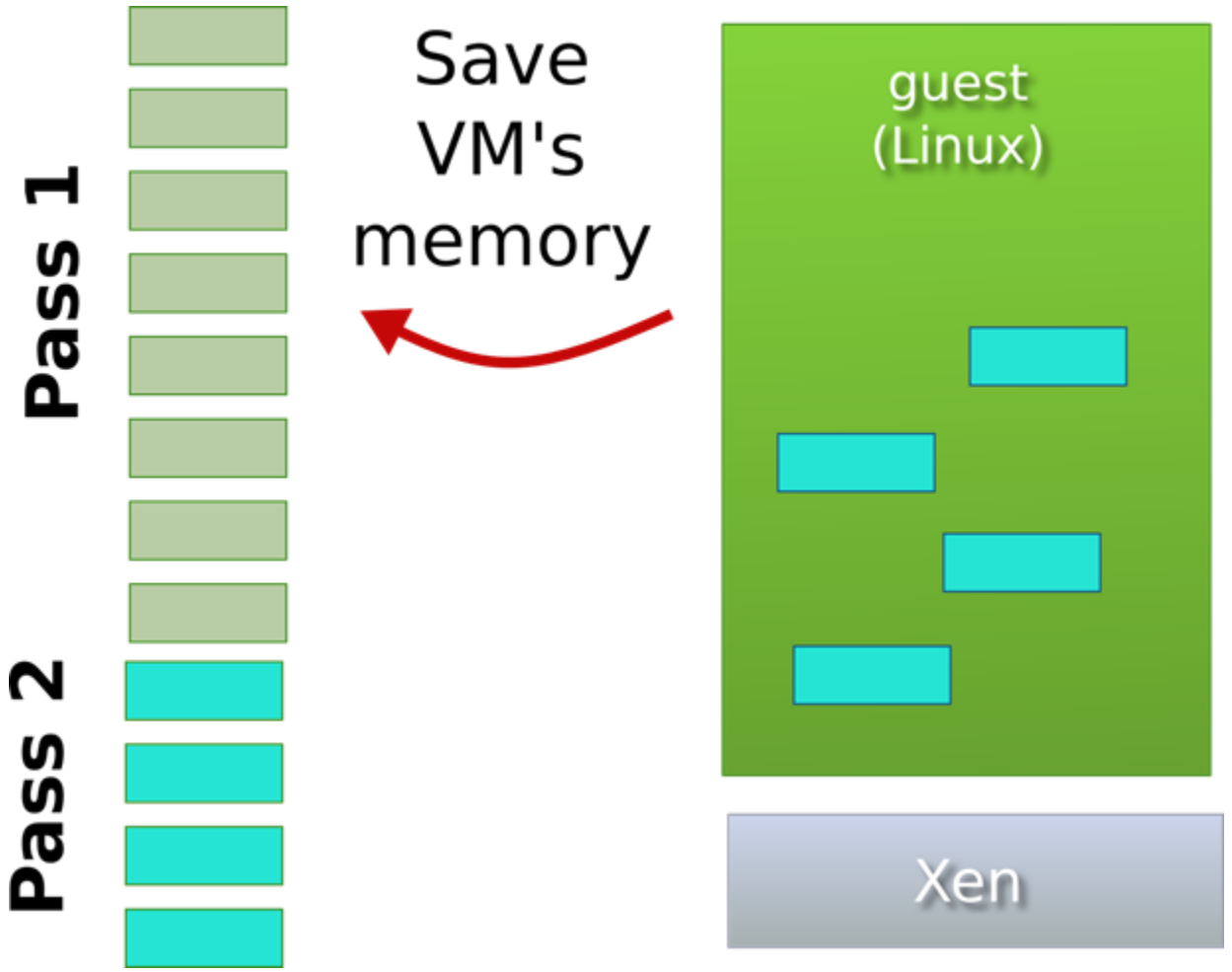
Migration: memory



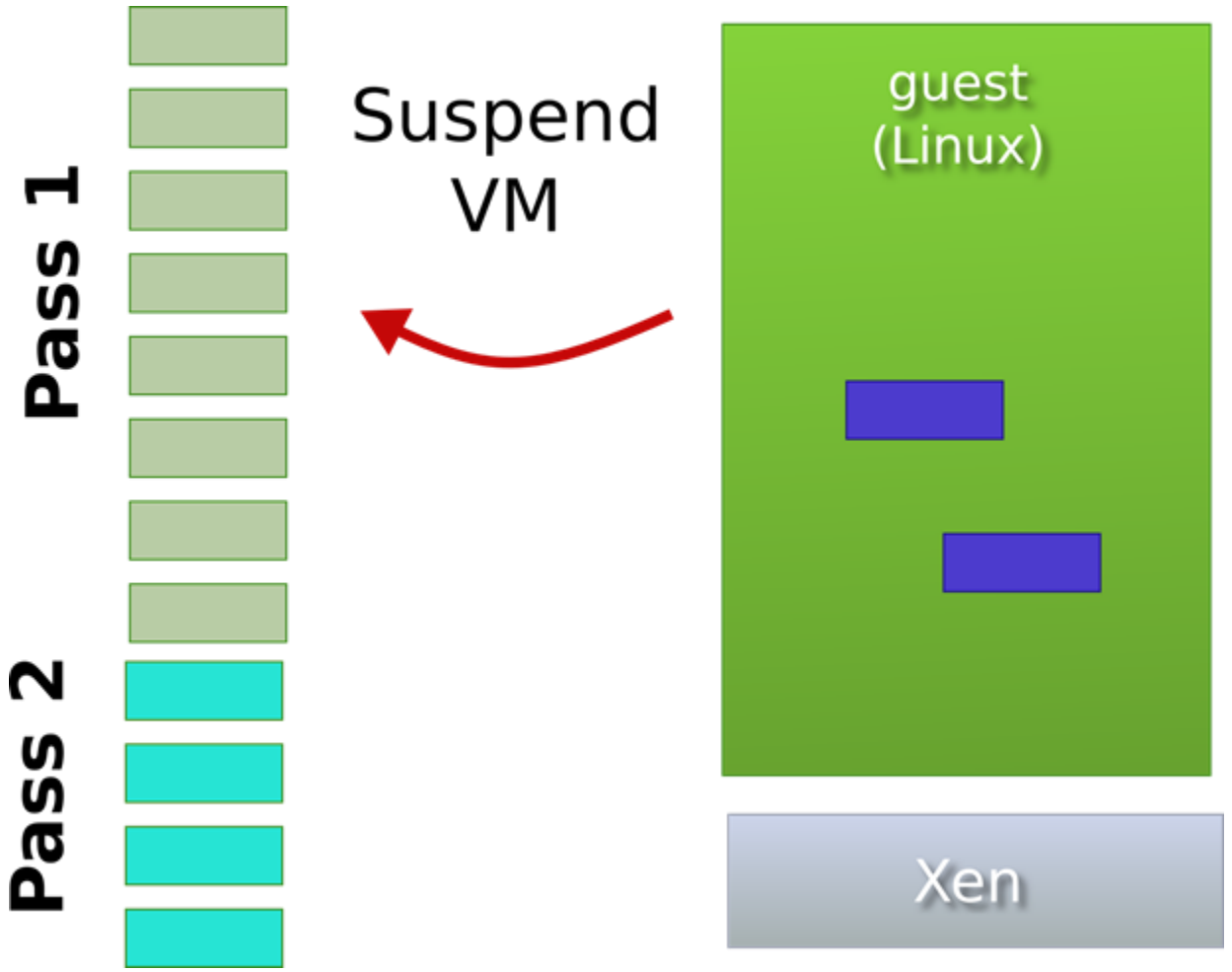
Migration: memory



Migration: memory



Migration: memory



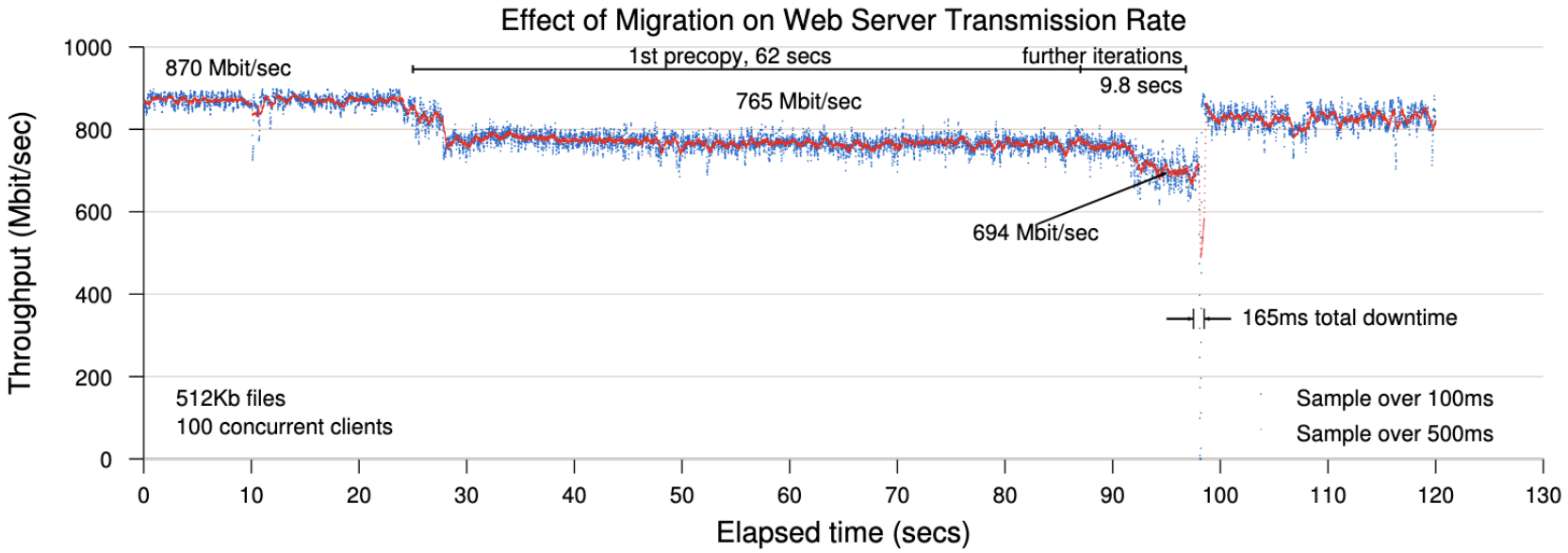


Figure 8: Results of migrating a running web server VM.

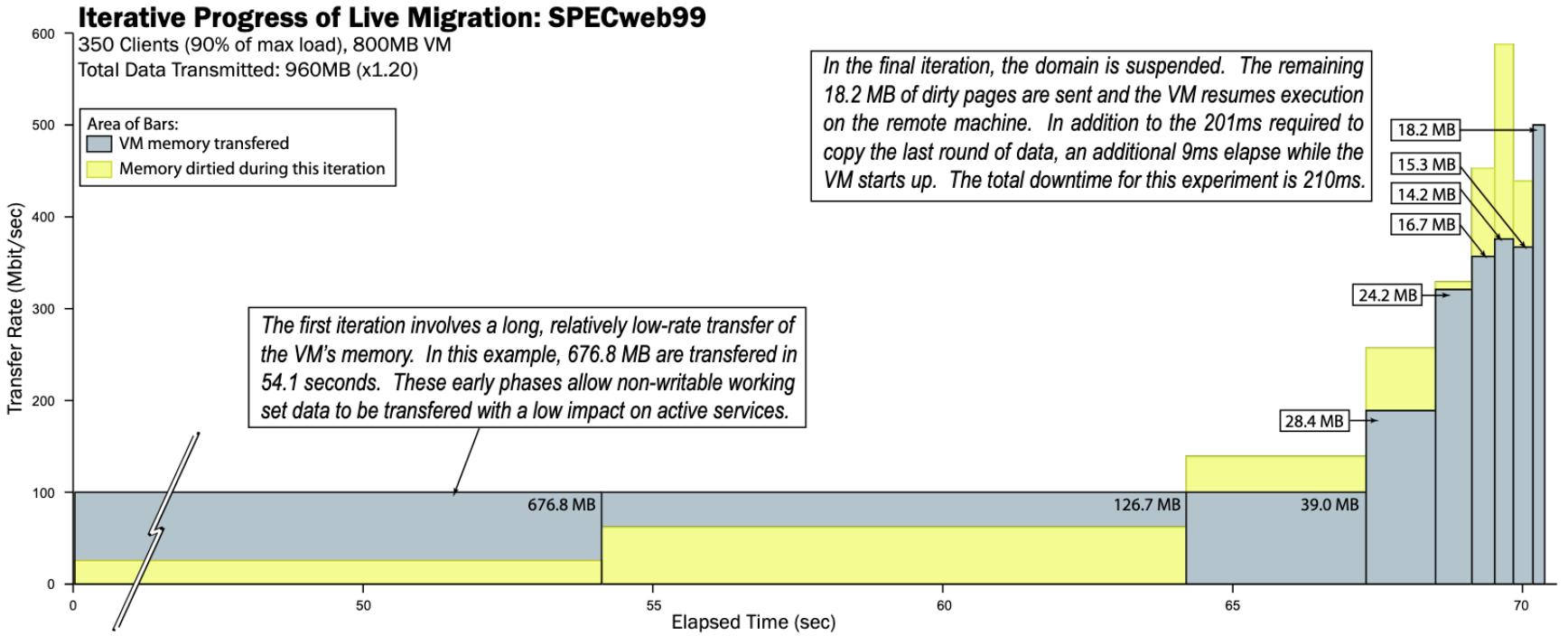


Figure 9: Results of migrating a running SPECweb VM.

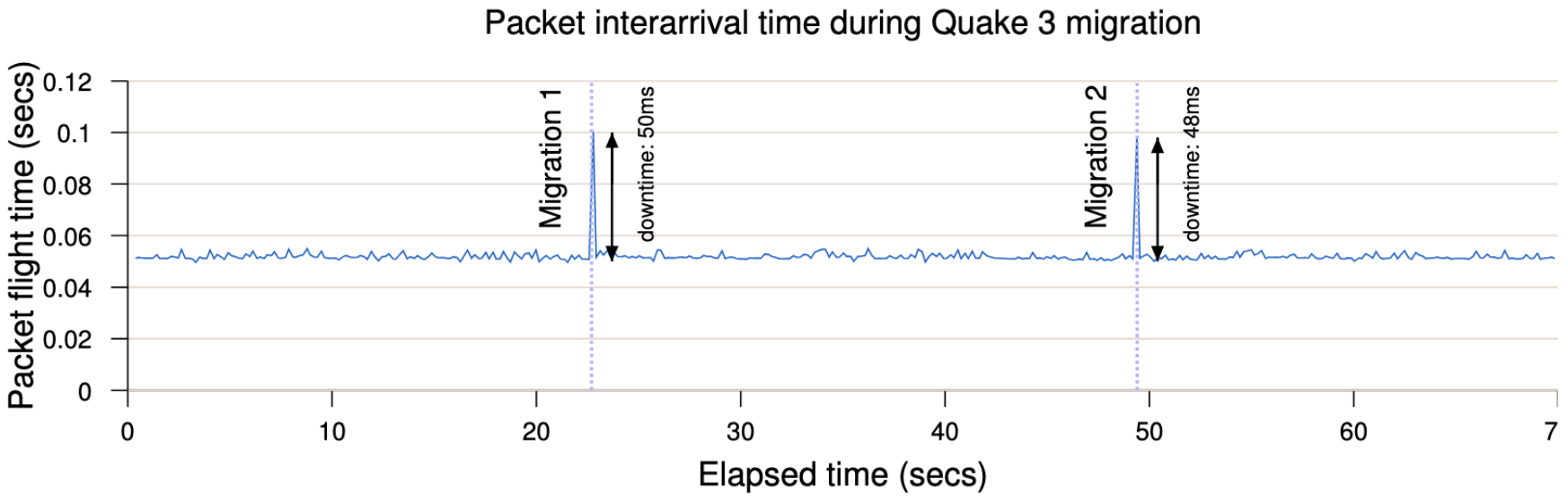


Figure 10: Effect on packet response time of migrating a running Quake 3 server VM.

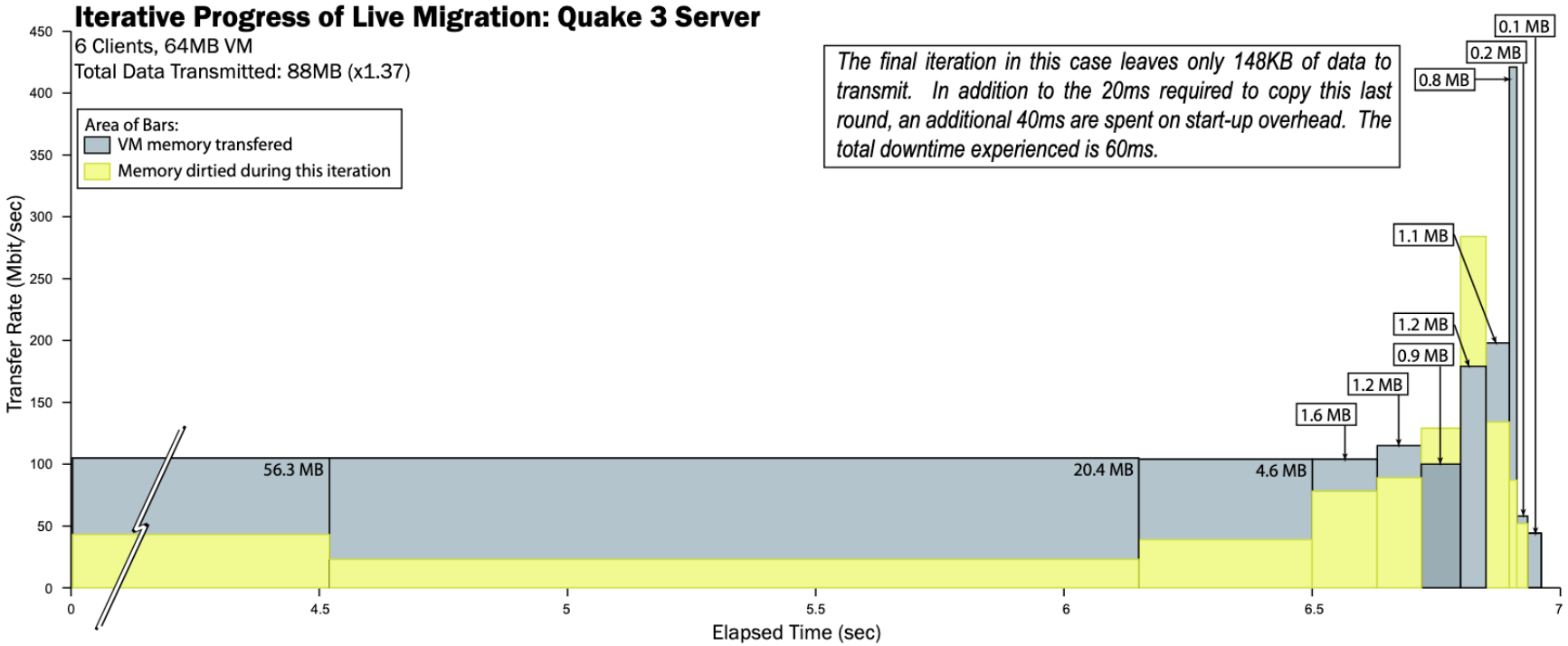
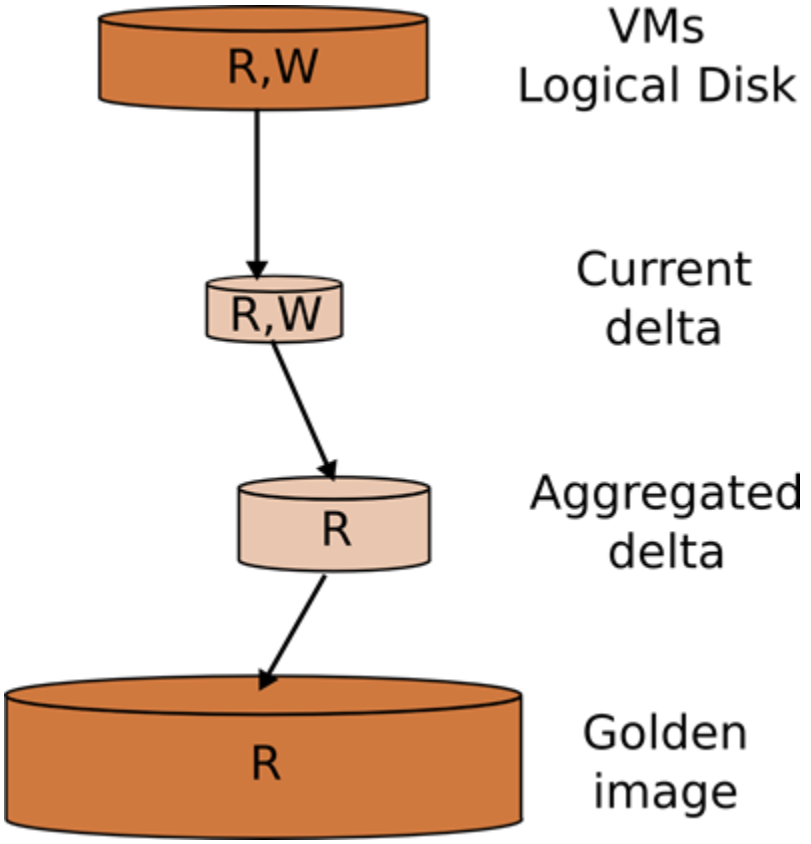
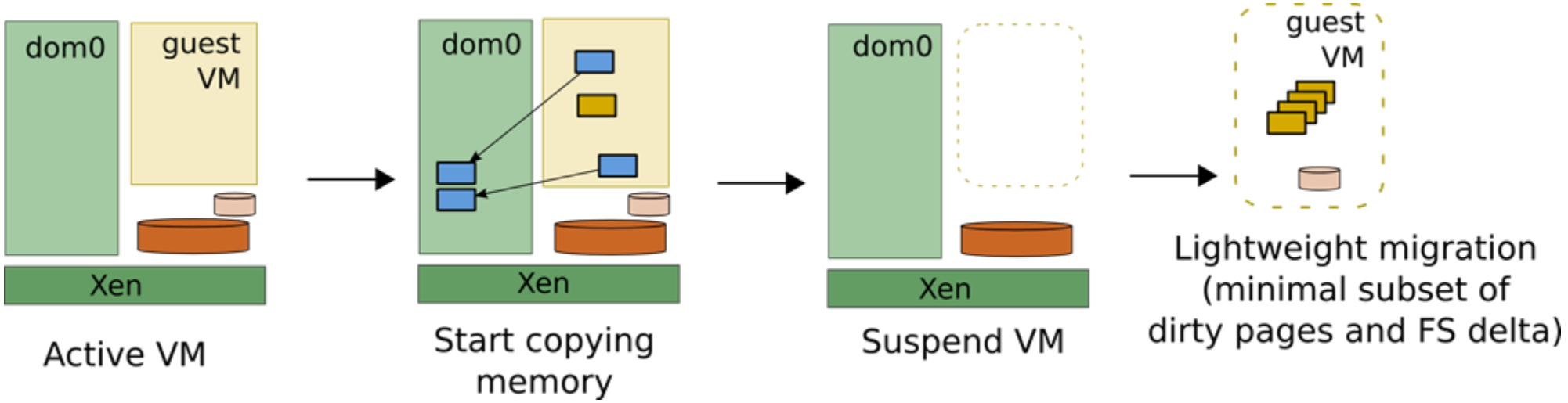


Figure 11: Results of migrating a running Quake 3 server VM.

Migration: storage



Migration


dom0
guest VM
Xen
Active VM
dom0
Xen
Start copying memory
dom0
Xen
Suspend VM
guest VM
Lightweight migration (minimal subset of dirty pages and FS delta)

Remus: High Availability via Asynchronous Virtual Machine Replication

Brendan Cully, Geoffrey Lefebvre, Dutch Meyer,
Mike Feeley, Norm Hutchinson, and Andrew Warfield*

Department of Computer Science
The University of British Columbia
{*brendan, geoffrey, dmeyer, feeley, norm, andy*}*@cs.ubc.ca*

Abstract

Allowing applications to survive hardware failure is an expensive undertaking, which generally involves re-engineering software to include complicated recovery logic as well as deploying special-purpose hardware; this represents a severe barrier to improving the dependability of large or legacy applications. We describe the construction of a general and transparent high availability service that allows existing, *unmodified* software to be protected from the failure of the physical machine on which it runs. *Remus* provides an extremely high degree of fault tolerance, to the point that a running system can transparently continue execution on an alternate physical host in the face of failure with only seconds of downtime, while completely preserving host state such as active network connections. Our approach encapsulates protected software in a virtual machine, asynchronously propagates changed state to a backup host at frequencies as high as forty times a second, and uses speculative execution to concurrently run the active VM slightly ahead of the replicated system state.

This paper describes *Remus*, a software system that provides OS- and application-agnostic high availability on commodity hardware. Our approach capitalizes on the ability of virtualization to migrate running VMs between physical hosts [6], and extends the technique to replicate snapshots of an entire running OS instance at *very* high frequencies — as often as every 25ms — between a pair of physical machines. Using this technique, our system discretizes the execution of a VM into a series of replicated snapshots. External output, specifically transmitted network packets, is not released until the system state that produced it has been replicated.

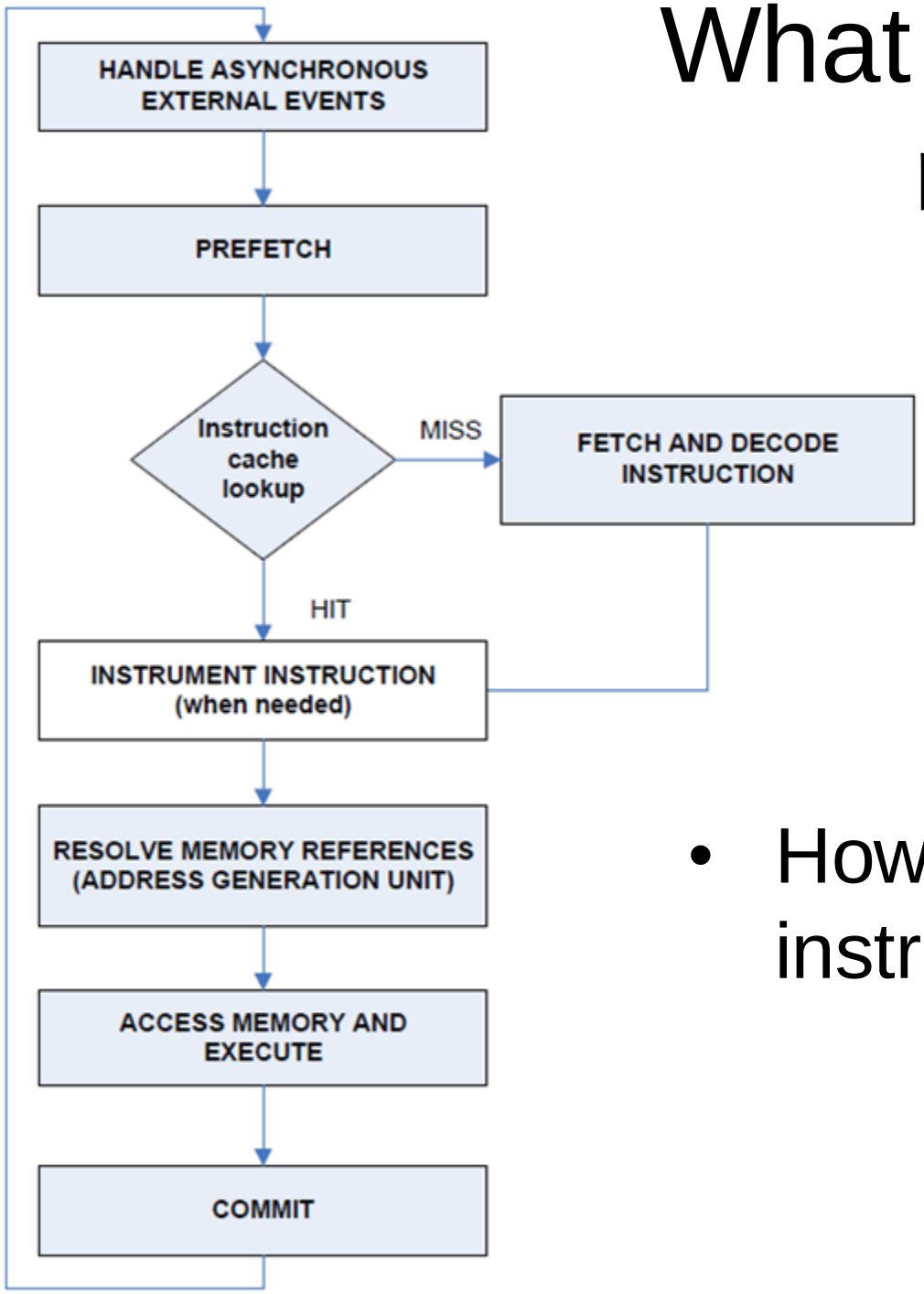
Virtualization makes it possible to create a copy of a running machine, but it does not guarantee that the process will be efficient. Propagating state synchronously at every change is impractical: it effectively reduces the throughput of memory to that of the network device performing replication. Rather than running two hosts in lock-step [4] we allow a single host to execute *speculatively* and then checkpoint and replicate its state *asynchronously*. System state is not made externally visible until the checkpoint is committed — we achieve high-

https://www.usenix.org/legacy/event/nsdi08/tech/full_papers/cully/cully.pdf

Remus

- Propagate checkpoints of active VM to a backup host
  - Combine live migration and low-overhead checkpointing

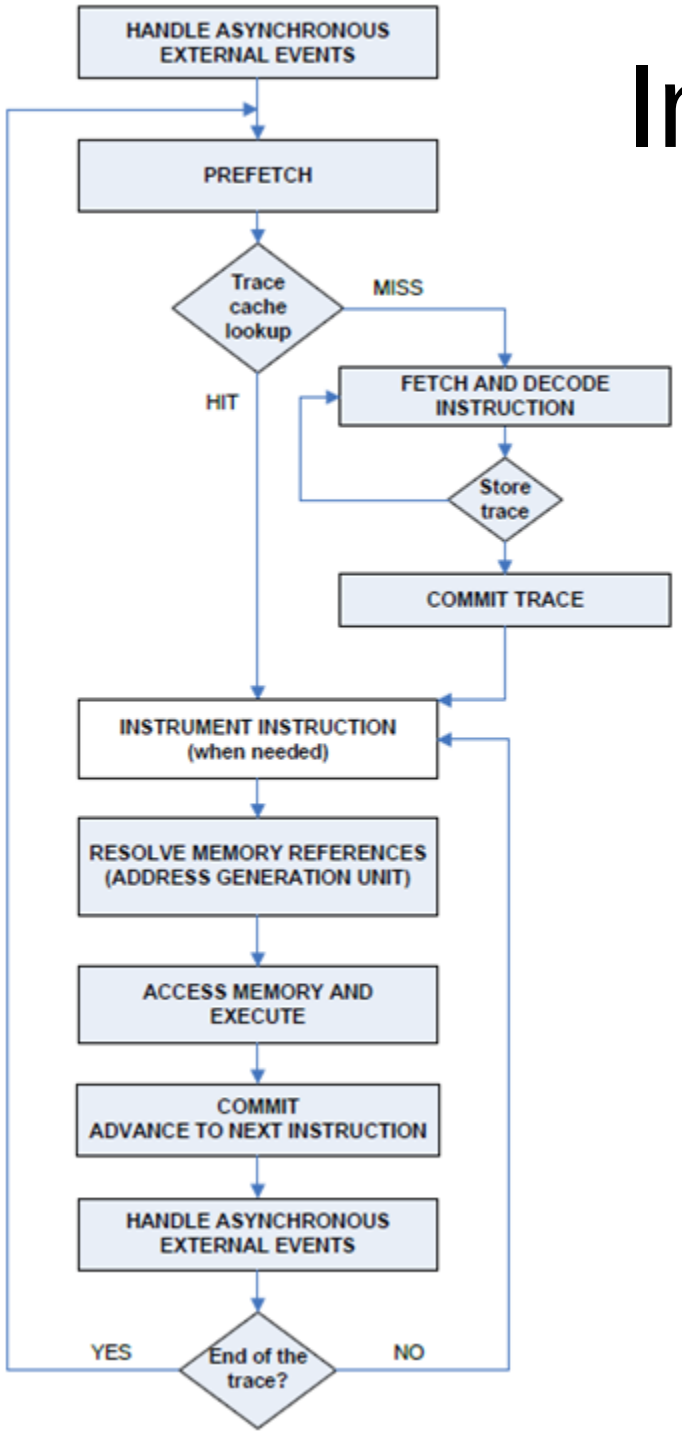
Interpreted execution revisited: Bochs



What does it mean to run guest?

- Bochs internal emulation loop
- Similar to non-pipelined CPU like 8086

- How many cycles per instruction?



Instruction trace cache

- 50% of time in the main loop
- Fetch, decode, dispatch
- Trace cache (Bochs v2.3.6)
- Hardware idea (Pentium 4)
- Trace of up to 16 instructions (32K entries)
- 20% speedup

Improve branch prediction

```
void BX_CPU_C::SUB_EdGd(bxInstruction_c *i)
{
  Bit32u op2_32, op1_32, diff_32;

  op2_32 = BX_READ_32BIT_REG(i->nnn());

  if (i->modC0()) {    // reg/reg format
    op1_32 = BX_READ_32BIT_REG(i->rm());
    diff_32 = op1_32 - op2_32;
    BX_WRITE_32BIT_REGZ(i->rm(), diff_32);
  }
  else {               // mem/reg format
    read_RMW_virtual_dword(i->seg(),
        RMAddr(i), &op1_32);
    diff_32 = op1_32 - op2_32;
    Write_RMW_virtual_dword(diff_32);
  }
  SET_LAZY_FLAGS_SUB32(op1_32, op2_32,
        diff_32);
}
```

- 20 cycles penalty on Core 2 Duo

Improve branch prediction

- Split handlers to avoid conditional logic
- Decide the handler at decode time (15% speedup)

Resolve memory references without misprediction

- Bochs v2.3.5 has 30 possible branch targets for the effective address computation
  - Effective Addr = (Base + Index*Scale + Displacement) mod(2^AddrSize)
  - e.g. Effective Addr = Base, Effective Addr = Displacement
- 100% chance of misprediction
- Two techniques to improve prediction:
  - Reduce the number of targets: leave only 2 forms
  - Replicate indirect branch point
- 40% speedup

Time to boot Windows

	1000 MHz Pentium III	2533 MHz Pentium 4	2666 MHz Core 2 Duo
Bochs 2.3.5	882	595	180
Bochs 2.3.6	609	533	157
Bochs 2.3.7	457	236	81

Cycle costs

	Bochs 2.3.5	Bochs 2.3.7	QEMU 0.9.0
Register move (MOV, MOVSX)	43	15	6
Register arithmetic (ADD, SBB)	64	25	6
Floating point multiply	1054	351	27
Memory store of constant	99	59	5
Pairs of memory load and store operations	193	98	14
Non-atomic read-modify-write	112	75	10
Indirect call through guest EAX register	190	109	197
VirtualProtect system call	126952	63476	22593
Page fault and handler	888666	380857	156823
Best case peak guest execution rate in MIPS	62	177	444

References

- A Comparison of Software and Hardware Techniques for x86 Virtualization. Keith Adams, Ole Agesen, ASPLOS'06
- Bringing Virtualization to the x86 Architecture with the Original VMware Workstation. Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, Edward Y. Wang, ACM TCS’12.
- Virtualization Without Direct Execution or Jitting: Designing a Portable Virtual Machine Infrastructure. Darek Mihocka, Stanislav Shwartsman, ISCA-35. https://bochs.sourceforge.io/Virtualization_Without_Hardware_Final.pdf

References

- Intel® 64 and IA-32 Architectures Software Developer's Manual. Volume 3C: System Programming Guide, Part 3
- Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. In ASPLOS'08.