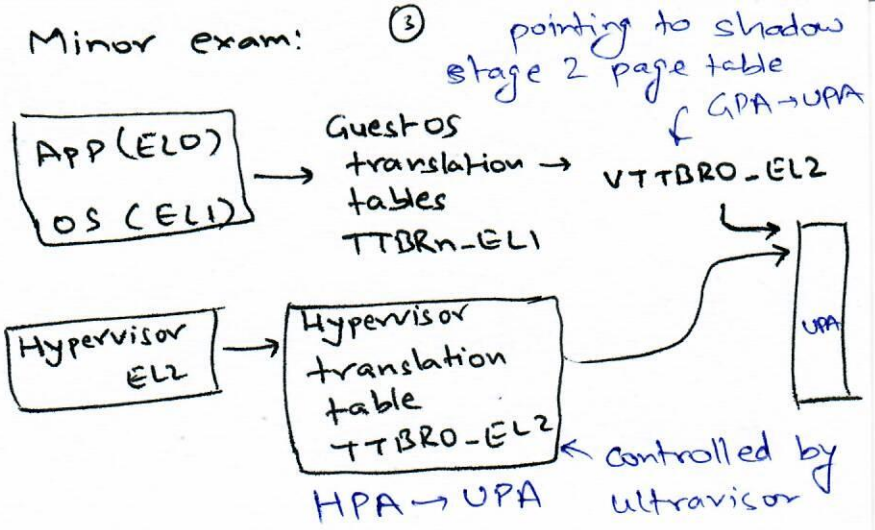
Minor exam: ③



③

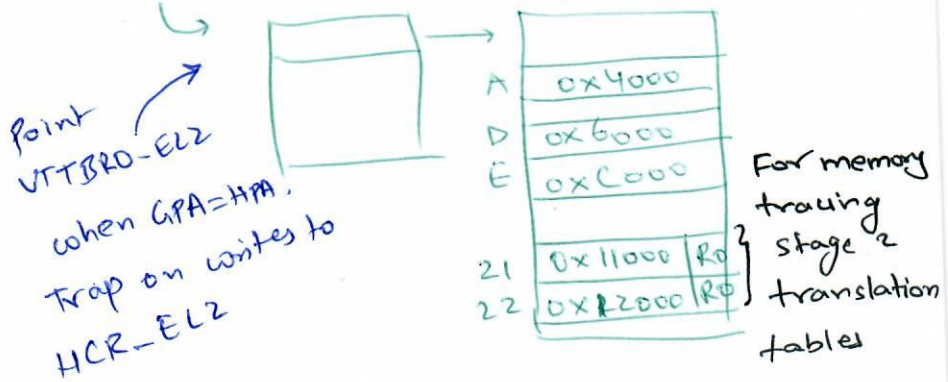
GVA	GPA	HPA	UPA
0x5000	0x1000	0xA000	0x4000
	0x7000	0xD000	0x6000
	0x8000	0xE000	0xC000
		0x21000	0x11000
		0x22000	0x12000

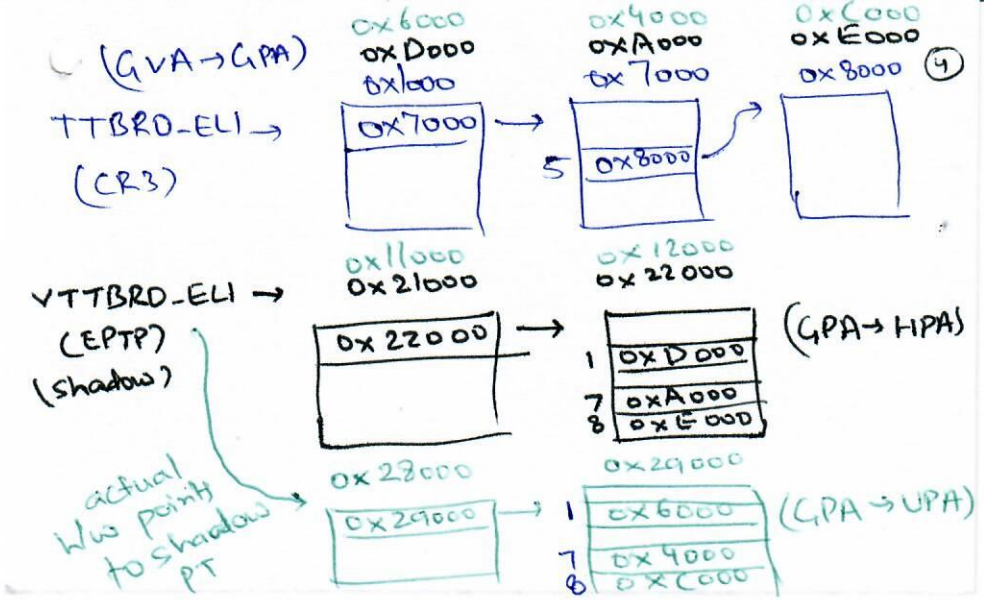
Goal:

0x5123 → 0xC123

②

TTBR0-EL2 → (HPA → UPA)





2-D address translation. (3)

0x5123? GVA

0x1000? → 0x28000 → 0x29000 → 0x6000

0x7000? → 0x28000 → 0x29000 → 0x4000

0x8000? → 0x28000 → 0x29000 → 0xC000

0x5123 → 0xC123

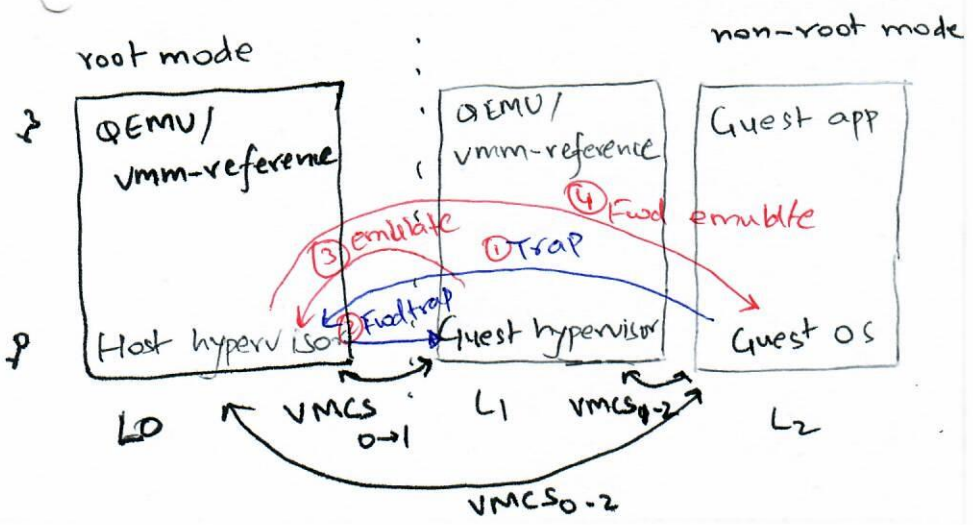
Pros: (2)

- Guest app fork is fast
- LD/ST of guest app is as fast as single level virtualizations

Traps

- TTBR0-EL2 and VTTBR0-EL2
  HCR_EL2

Nested virtualization on x86 (3)



---

(2)

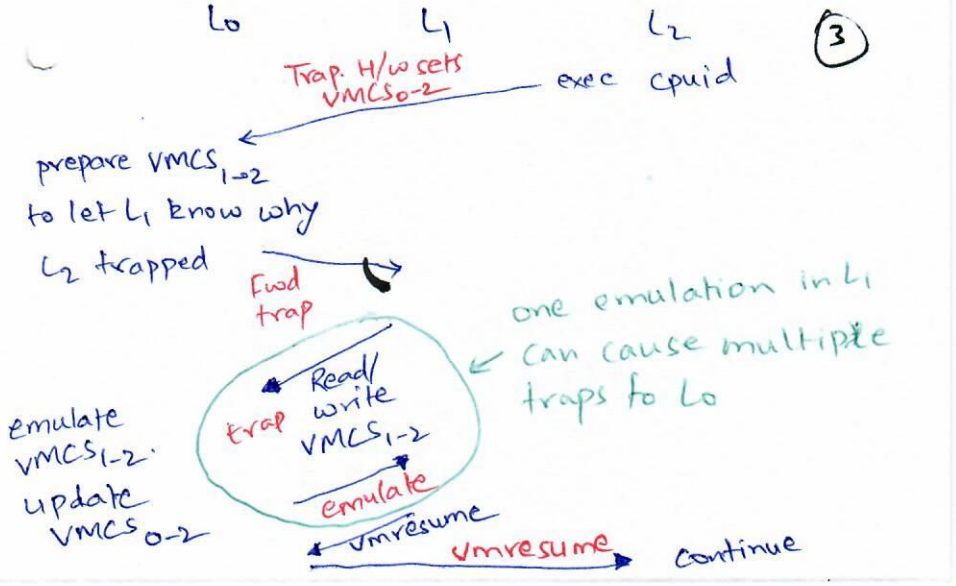
Similar to how we shadow EPT_{1-2} to point h/w to EPT_{0-2}

we shadow $VMCS_{1-2}$ to point h/w to ~~&~~ $VMCS_{0-2}$

H/W is never pointed to EPT_{1-2} / $VMCS_{1-2}$

---

(3)



(3)

Emulation overhead ∝ Traps from L1 to L0

PIO exit → 31 exits → 14 exits } AMD
cpuid → 13 exits → 2 exits
↳ 80% less overhead in AMD

AMD could let modification to VMCS through LD/ST. Intel needs privileged instructions: vmread/vmwrite
Trap for L1 to L0

---

(1)

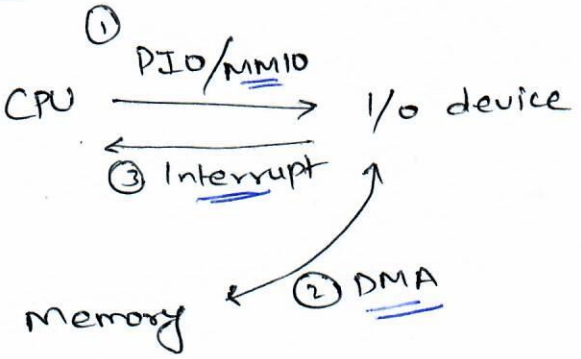
- 6-8% overhead over single-level hypervisor
- Available in Kvm. nested=true
- More in Turtles [OSDI'10] (best paper award!)

I/O virtualization goals: (2)

* Safety - Isolation
* Performance - closest to native
* Transparency - unmodified Guest

User space	Application
Guest OS	Device driver
Hypervisor	Virtual device ~> Emulated

Background → (1)

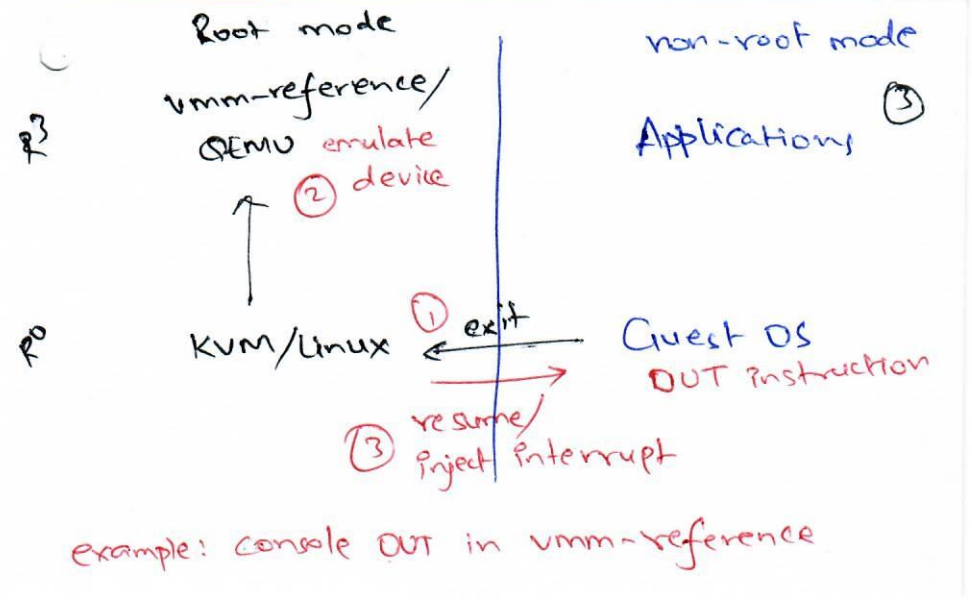


Port I/O (2)

IN AL, DX	1 byte	OUT DX, AL
IN AX, DX	2 bytes	OUT DX, AX
IN EAX, DX	4 bytes	OUT DX, EAX

DX: 0x6000 - 0x6004 keyboard/mice PS/2 connected

0x03f8 console



example: console OUT in vmm-reference

Memory mapped I/O — LD/ST instructions

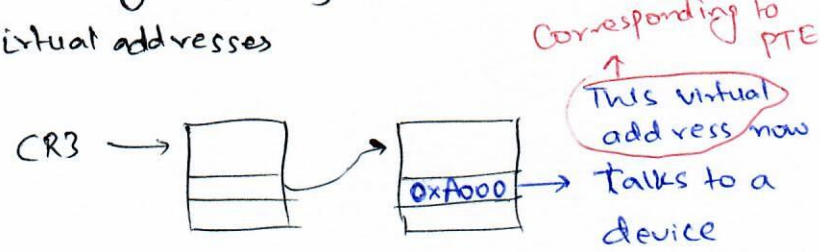
Address range	Size	Device
0000 - 7FFF	32 kB	RAM
8000 - 8DFF	256 bytes	General I/O
9000 - 90FF	256 bytes	Sound controller
A000 - A7FF	2 kB	Video controller

MMIO BAR base address register for video controller

Sample system memory map

Managed by hardware's "address decoding circuitry"

Memory map is of PHYSICAL ADDRESSES, not virtual addresses



Memory mapped I/O

(3)

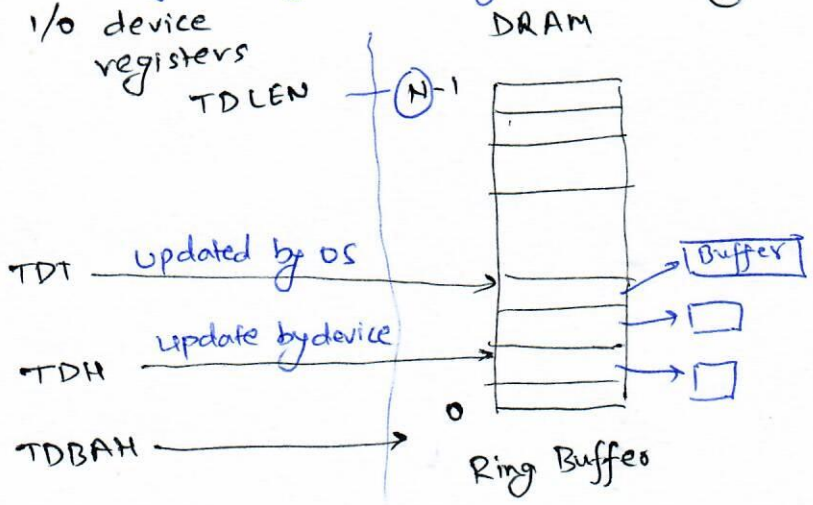
Example: e1000 (BAR: 0xF0000)

Table 6.1 book

name	offset	purpose
STATUS	0x00008	device states
ICR	0x000C0	bitmap of causes
IMS	0x000D0	enable interrupts
IMC	0x000D8	disable "

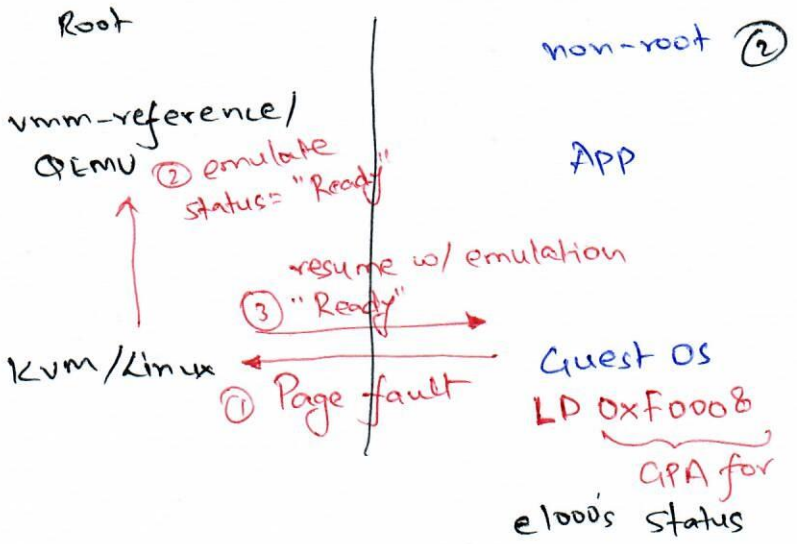
e1000 register for Tx ring

(3)

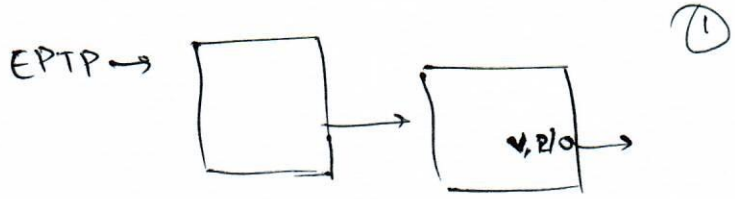


(1)

name	offset	purpose
TDBAH	0x3800	base address of Tx ring
TDLEN	0x3808	Tx ring size
TDH	0x3810	ptr to head of ring
TDT	0x3818	" " tail " "



Memory tracing: getting page faults



- Unset valid bit: get R/W faults
- Unset R/O bit: get W faults.

I/O emulation

Transparency ✓

```
uint8_t mac_icr_read(E1000State *s) {   ②
    uint32_t r = s->mac_reg[ICR];
    s->mac_reg[ICR] = 0;
    return r;
}
```

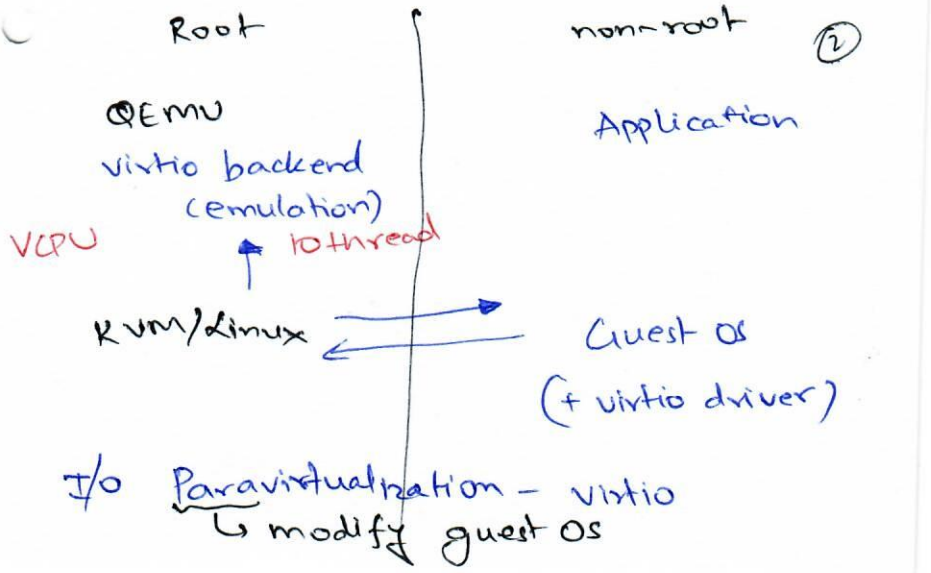
code from QEMU

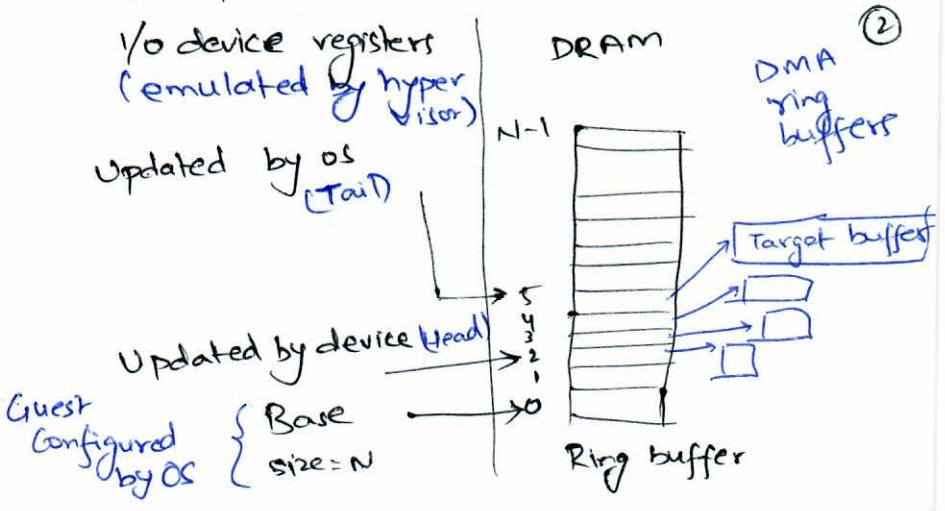
↑ emulate the behavior that real hardware clears ICR when reading

Problem → e1000 (3)

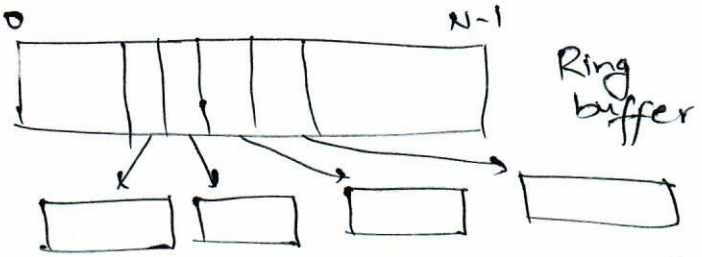
Physical devices were not designed for virtualization.

- Every packet send → Read status twice (VM exits)
- Mark status R/O ?
- Status, ICR, IMS, IMC are on same page ∴ must be read protected



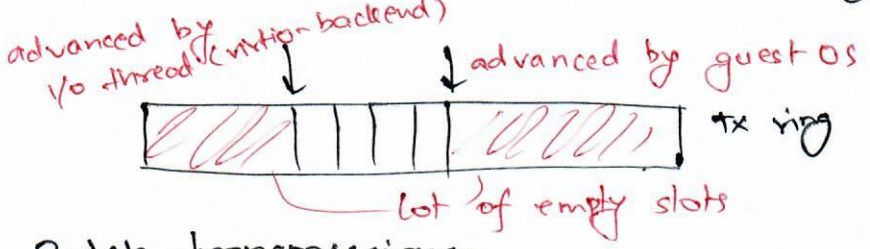


Optimizations - virtqueue_kick (2)



- 4 exits in emulated full I/O virtualization
- 1 exit. write batch, then call virtqueue_kick

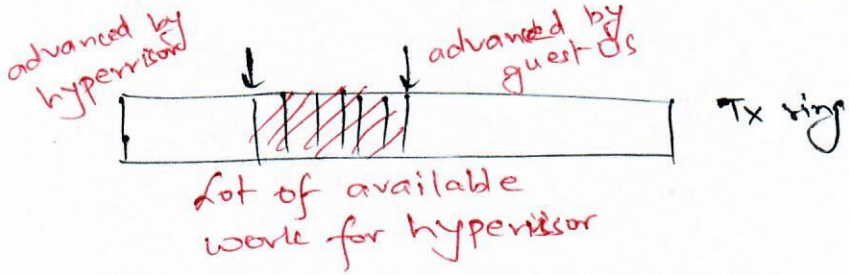
Optimizations - NO_INTERRUPT (2)



Batch transmissions-

- Don't interrupt guest OS after servicing a buffer
- Guest OS disables NO_INTERRUPT if Tx ring is full

Optimization - NO_NOTIFY (2)



Hypervisor: I am already working on it. No need to kick me again.

Table 6-2: netperf benchmark virtio/e1000 (2)

Guest OS		
	Throughput	22x
	Exits/sec	1/30x
	Interrupts/sec	1/14x

Per TCP segment send:

e1000: 9 exits (not virt as design goal)

virtio: 1/25 exits (batching, explicit kicks, NO_NOTIFY)

Per TCP segment send (2)

e1000

1 interrupt per segment

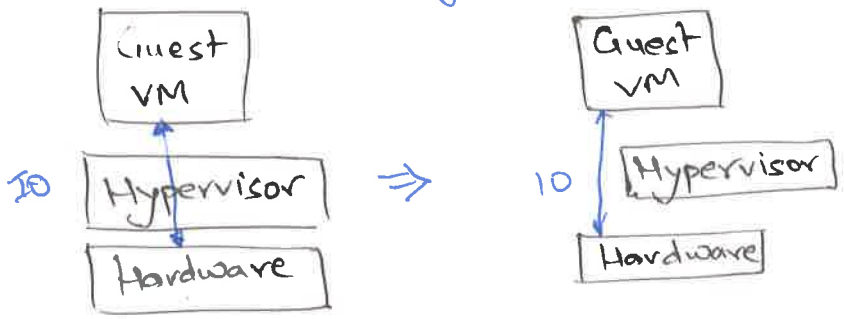
virtio

1 interrupt per 118 segments

↓

NO-INTERRUPT flag

Direct device assignment ①



Problems

1. Scalability: Number of VMs > I/O devices → SRIOV

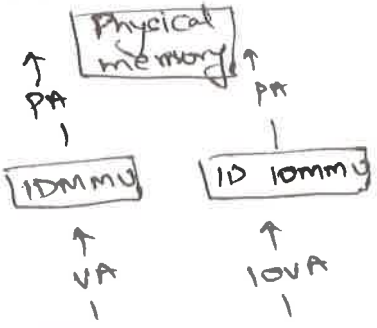
2. Safety: Guest may ask device to read write any physical memory → IPMMU

3. VMs do not know physical memory addresses for DMA requests ①

4. Hypervisor has to pin all the guest pages (any page can be DMA targets) ⇒ No memory swapping

IOMMU

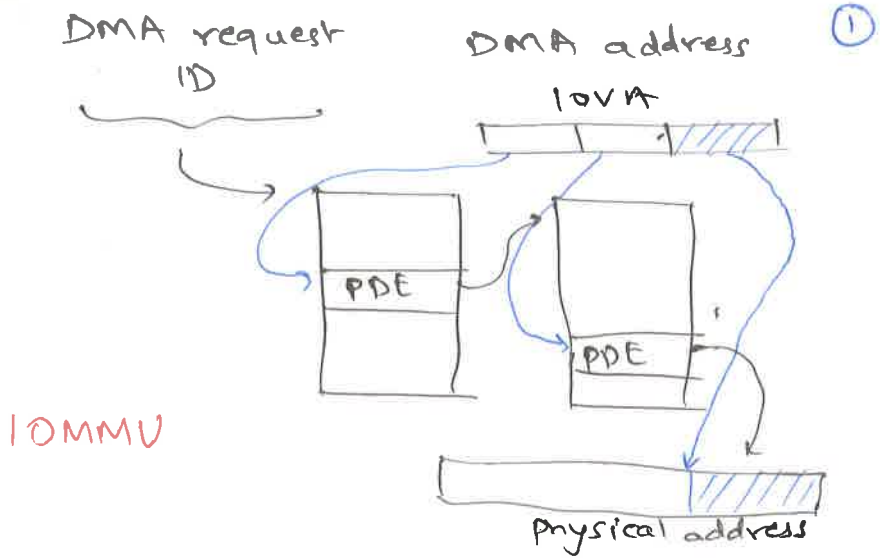
IOMMU - Intel VT-d



- OS ~~Guest~~ doesn't want to let IO device read/write entire physical memory

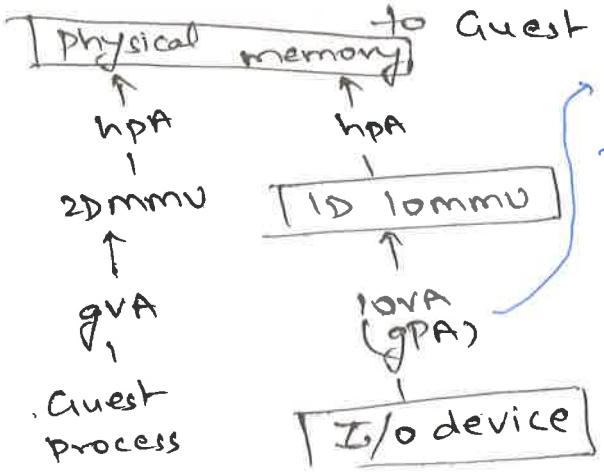
⇒ Control access w/ IOMMU

- Bare metal 1-D I/O MMU



Virtualization - Don't advertise IOMMU to Guest

2



Guest thinks there is no IOMMU.

Puts gPA in requests

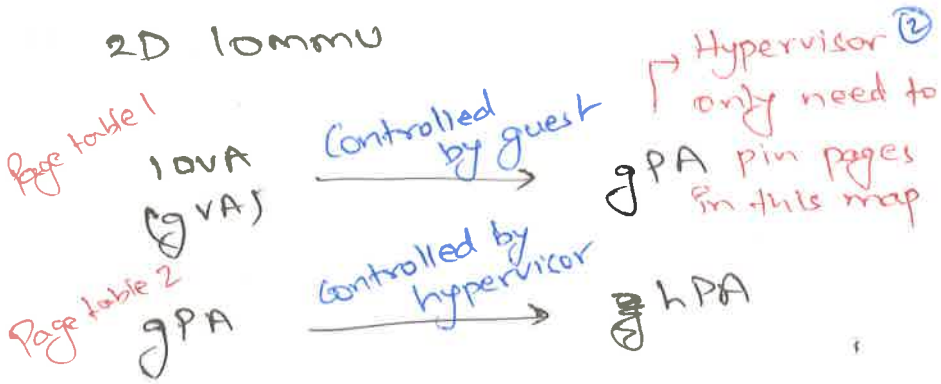
1 D Iommu ≡ EPT ①

Controlled by hypervisor

1. Hypervisor can still not swap guest pages
2. Buggy device can write any guest memory

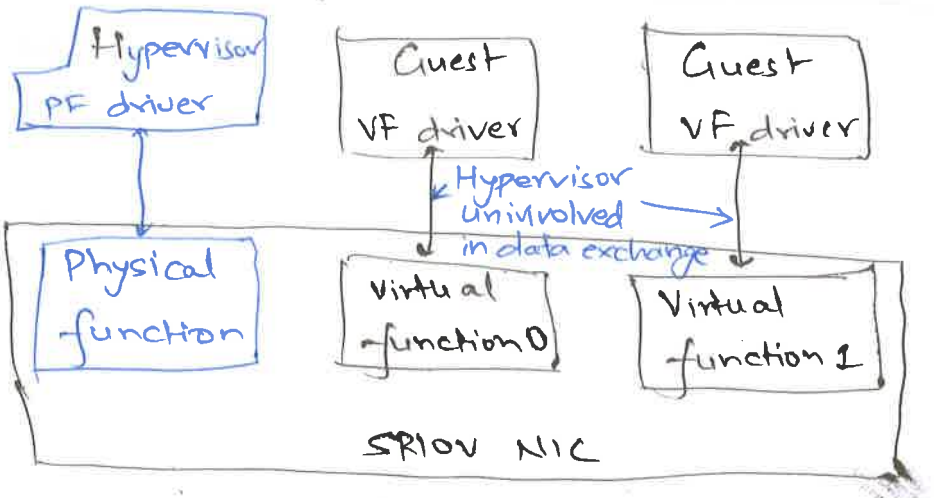
Isolation ✓

2D Iommu



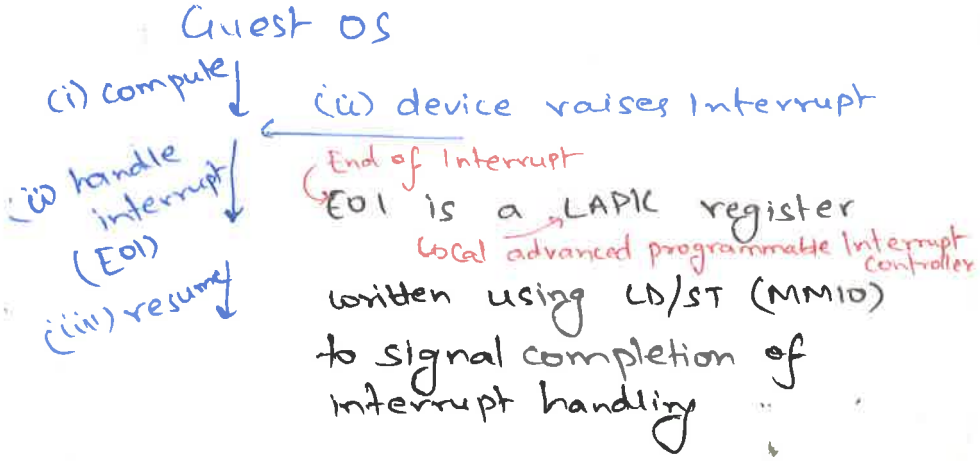
② Hypervisor only need to pin pages in this map

- Guest protected from buggy devices



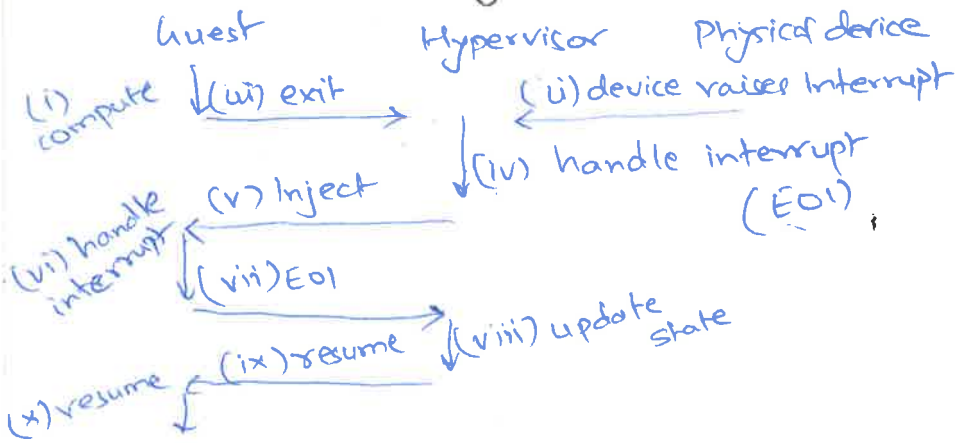
1.31 x - 1.78x faster than virtio

Interrupt delivery (OS view) ①



~~Single root I/O virtualization~~ ②

Interrupt delivery



② (v) Inject interrupt - Write to the "interrupt information" field of VMCS. Interrupt delivered in next VMENTRY.

(iv, vii) EOI - LAPIC register to signal EOI

↳ can be skipped with new hardware

x2APIC → same idea as e1000. Previously, LAPIC emulated by memory tracing. x2APIC uses separate read MSR/write MSR instructions

Live migration

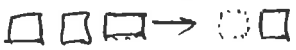
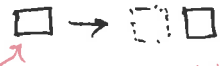
(2)

Goals

- minimal downtime: Interactive clients playing a game
- minimal total migration time
- don't exhaust n/w b/w → maintain QoS

why live migration?

(1)

- Maintenance: eg. kernel update of server 1
  - HPC labs: servers not restarted for years
- Need multi-party coordination
- Long running sciences jobs
- Load balancing 
- Consolidation 

shutdown, save energy bills

What needs to migrate?

(1)

- Assume network attached storage
- Can carry IP, clients connections
  - Broadcast an ARP packet containing new MAC address <> IP mapping
- CPU state, memory contents

Basic approach: Stop and copy ②

All guest pages are in EPT
- copy all guest pages

Copy over VMCS
- copies paused guest's CPU state

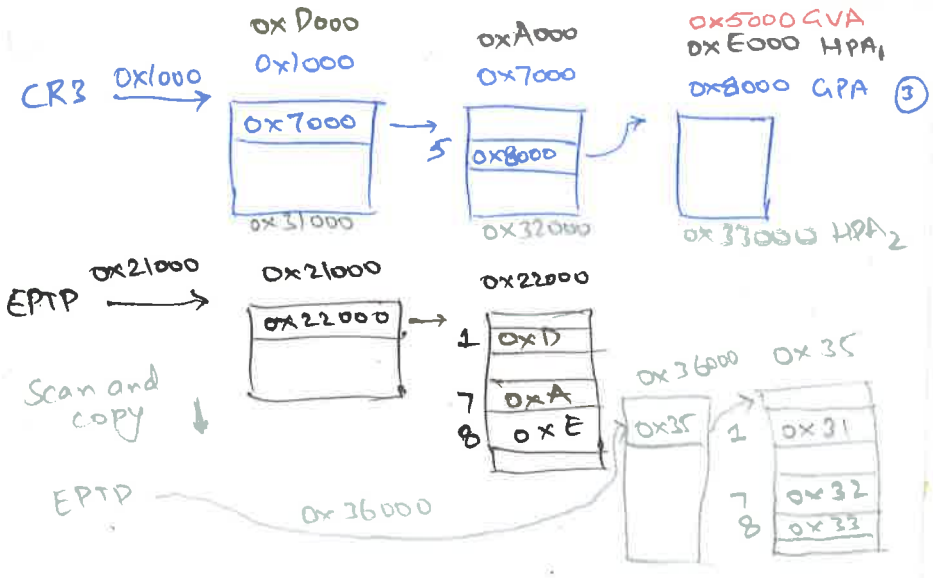
Update EPTP. Update IP routing

VMENTRY

②

GVA	GPA	HPA_1	HPA_2
	0x1000 →	0xD000 →	0x31000
	0x7000 →	0xA000 →	0x32000
0x5000 →	0x8000 →	0xE000 →	0x33000

We copy all guest's pages from H_1 to H_2



New page walk GVA 0x5123 (2)

0x1000? → 0x36000 → 0x35000 → 0x31000

0x7000? → 0x36000 → 0x35000 → 0x37000

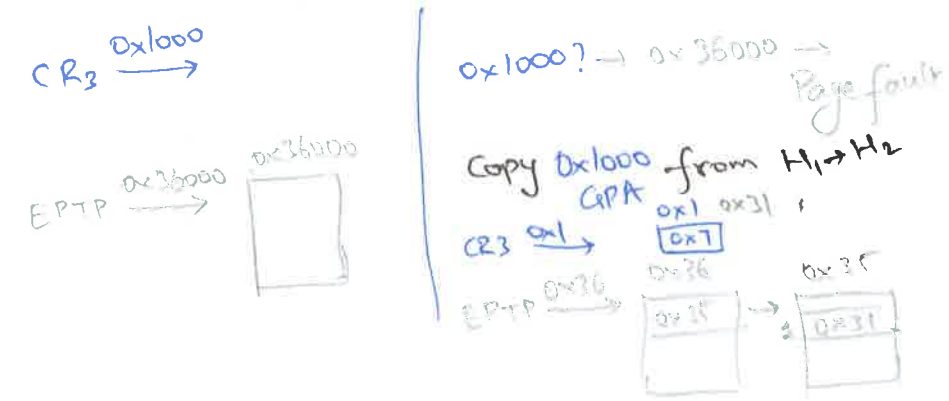
0x8000? → 0x36000 → 0x35000 → 0x33000

Problem: Large downtime ≈ $\frac{\text{Guest memory}}{\text{N/w bandwidth}}$

Idea: Just copy VMCS and resume (3)

CPU state

GVA 0x5123



(3)

0x1000? → 0x36 → 0x35 → 0x31

0x7000? → 0x36 → 0x35 → Page fault

- Again copy 0x7000 GPA from $H_1 \rightarrow H_2$
- Accordingly update EPT on H_2.
- Repeat for every page fault

"Post copy" ①

Best downtime ≈ n/w latency
↑ Only copying VMCS in stop and copy phase

Reality: Terrible QoS after migration.
Service feels down
Network download on every page-fault

Optimization: Hide network tranfers ③

Best case: Tranfer the page just before it is accessed.

Bubbling w/ R pivots (assume locality in GPA access)

Last R GPA download requests recieved from H_2

R pivots → Guest may be running multiple processes!

Grow the bubble around a pivot

⇒ Proactively push pages from $H_1 \rightarrow H_2$

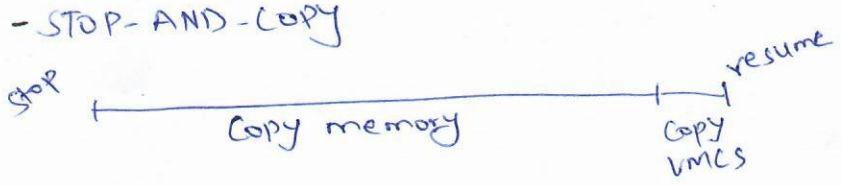
①

- Still 17-57% of page faults are n/w page faults (Figure 14, post copy)

- Dual multipivot bubbling reduces n/w page fault (Figure 17, post copy)

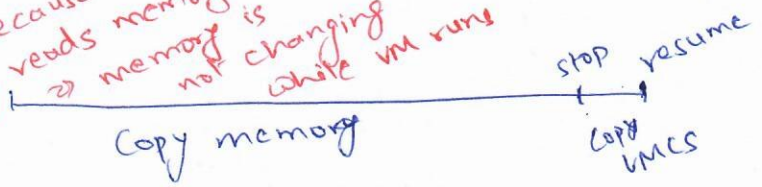
What if VM only READS memory? ②

- STOP-AND-COPY



PRE COPY

OK because VM only reads memory ⇒ memory is not changing while VM runs

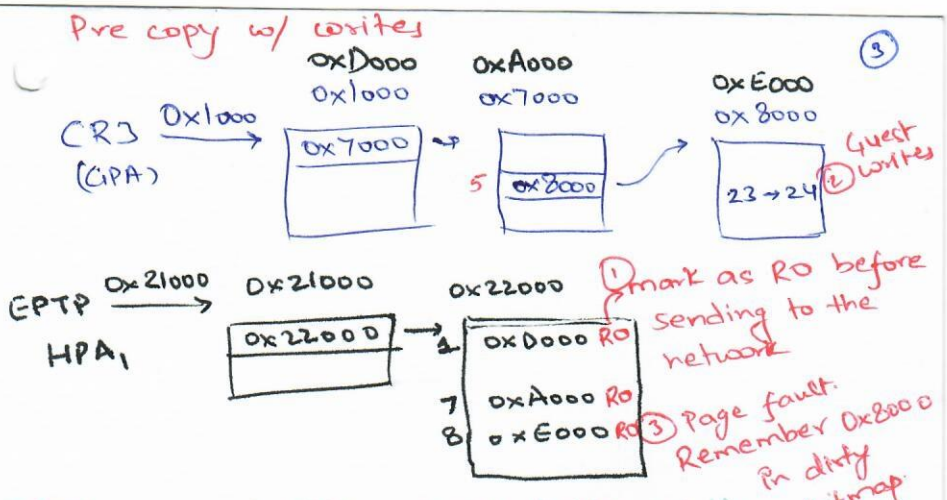


Benefits ①

- Same downtime as post-copy
  while copying VMCS
- No n/w page faults
  Excellent service after migration

Can not handle writes ∴

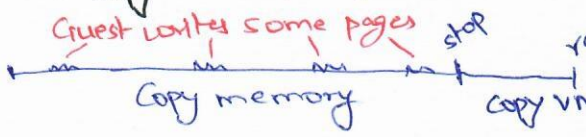
Pre copy w/ writes ③

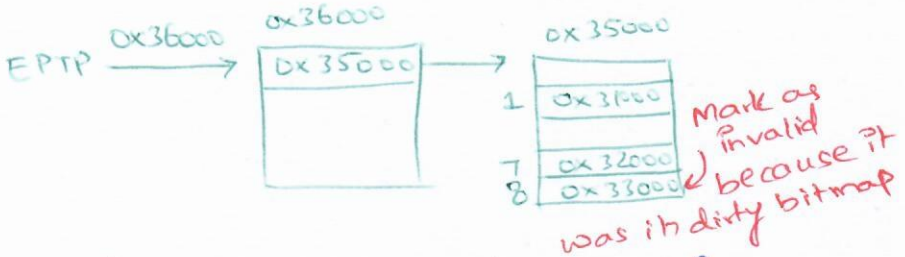


EPTP ----> building page table as it recieves pages
HPA2

Pre copy with writes

(2)





LD GVA 0x5132 cause n/w page fault.

Downtime - minimal

(2)

N/w usage - higher. dirty pages have to be transferred twice.

QoS after migration - Better than pure post copy. No n/w fault for read only pages

=> Depends on writable working set size

Writeable working set (WWS)

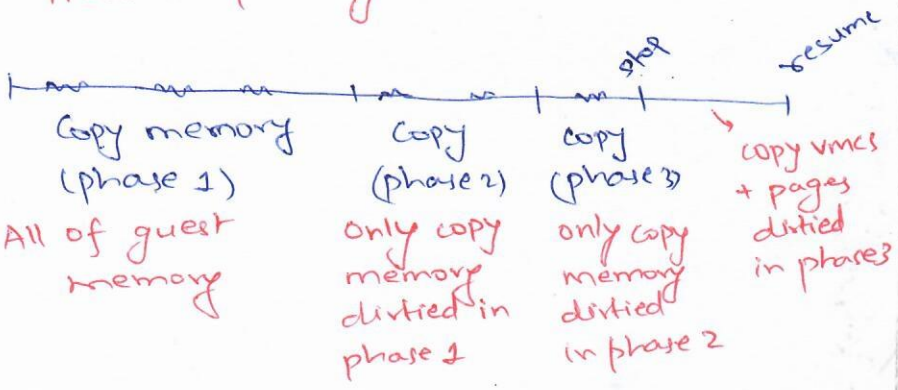
(1)

number of pages dirtied during the pre copy phase

High WWS => More n/w usage,
more n/w page faults
after migration

Improve QoS after migration - (2)

Iterative pre copy



Reasoning - (1)

Phase 1 copied more memory ⇒ Longer
⇒ More dirty pages

Phase 2 copied only phase 1 dirty page

⇒ Each phase gets shorter, copies lesser and lesser pages

Tradeoffs (1)

- Excellent QoS after migration
  No n/w page faults
- Uses more n/w
  Same page may be transfered many times
- Large WWS causes large downtime

- Migrate game server with 60ms downtime! (Live migration paper)

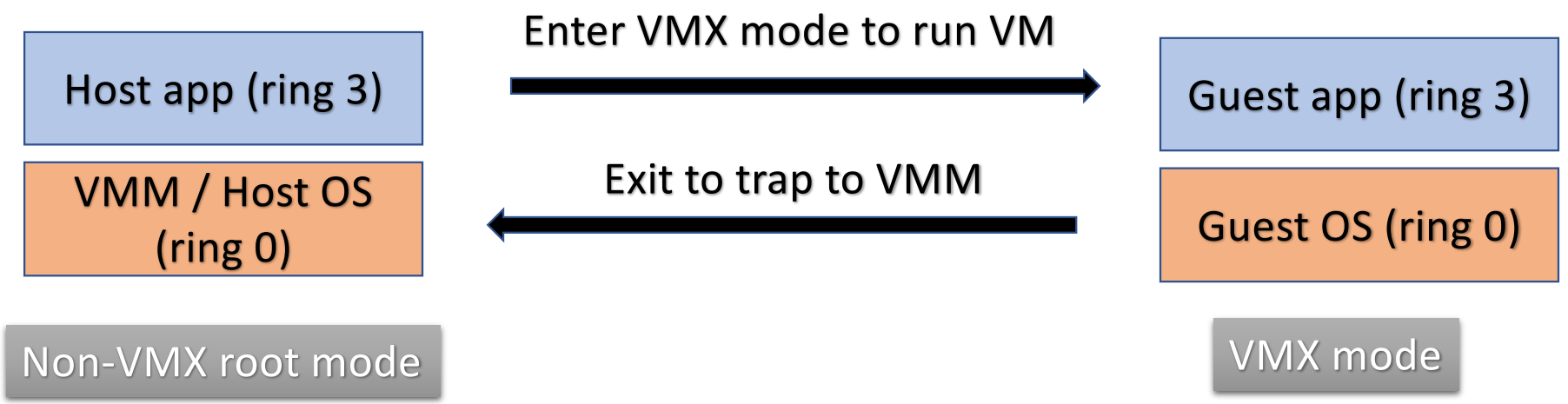
- QEMU, XEN implement iterative pre copy
  - User visible downtime more important than n/w usage!

Virtualization without Architectural support

Some slides borrowed from Mythilli, IITB

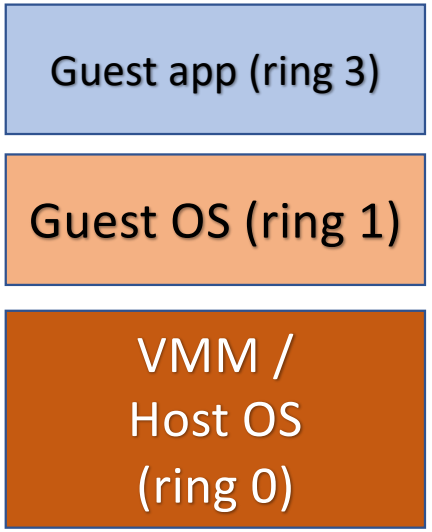
Hardware-assisted CPU virtualization

- Example: KVM/QEMU in Linux
  - CPU has a special VMX mode of execution
  - x86 has 4 rings on non-VMX root mode, another 4 rings in VMX mode
- VMM enters VMX mode to run guest OS in (special) ring 0
- Exit back to VMM on triggers (VMM retains control)



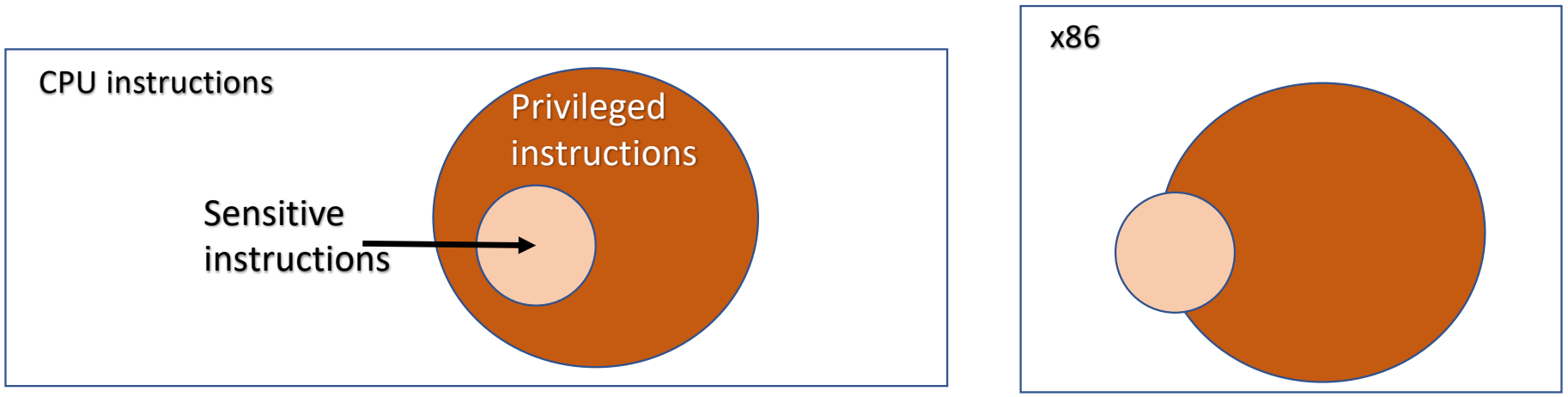
Trap and emulate VMM on older x86

- All CPUs have multiple privilege levels
  - Ring 0,1,2,3 in x86 CPUs
- Normally, user process in ring 3, OS in ring 0
  - Privileged instructions only run in ring 0
- Now, user process in ring 3, VMM/host OS in ring 0
  - Guest OS must be protected from guest apps
  - But not fully privileged like host OS/VMM
  - Can run in ring 1?
- Trap-and-emulate VMM: guest OS runs at lower privilege level than VMM, traps to VMM for privileged operation



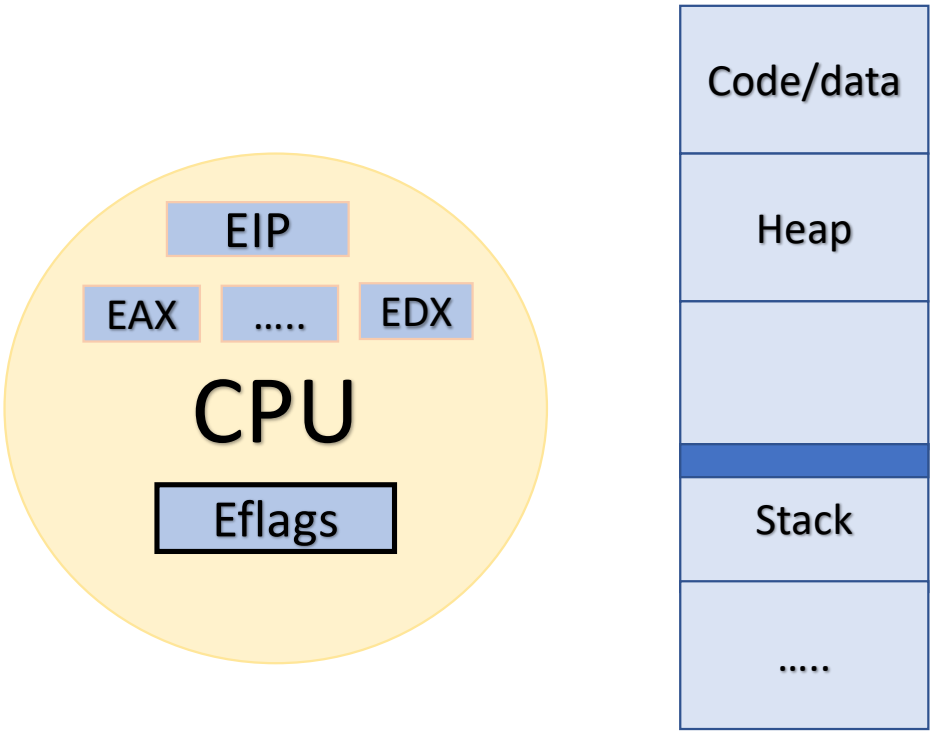
Popek Goldberg theorem

- Sensitive instruction = changes hardware state
- Privileged instruction = runs only in privileged mode
  - Traps to ring 0 if executed from unprivileged rings
- In order to build a VMM efficiently via trap-and-emulate method, sensitive instructions should be a subset of privileged instructions
  - x86 does not satisfy this criteria, so trap and emulate VMM is not possible



x86 does not follow Popek-Goldberg theorem

- Eflags register is a set of CPU flags
  - IF (interrupt flag) indicates if interrupts on/off
- Consider the popf instruction in x86
  - Pops values on top of stack and sets eflags
- Executed in ring 0, all flags set normally
- Executed in ring 1, only some flags set
  - IF is not set as it is privileged flag
- So, popf is
  - behavior sensitive: behaves differently when executed in different privilege levels
  - Not privileged, does not trap
  - => Guest OS is buggy in ring 1



x86 does not follow Popek-Goldberg theorem (2)

Table 2.2: List of sensitive, unprivileged x86 instructions

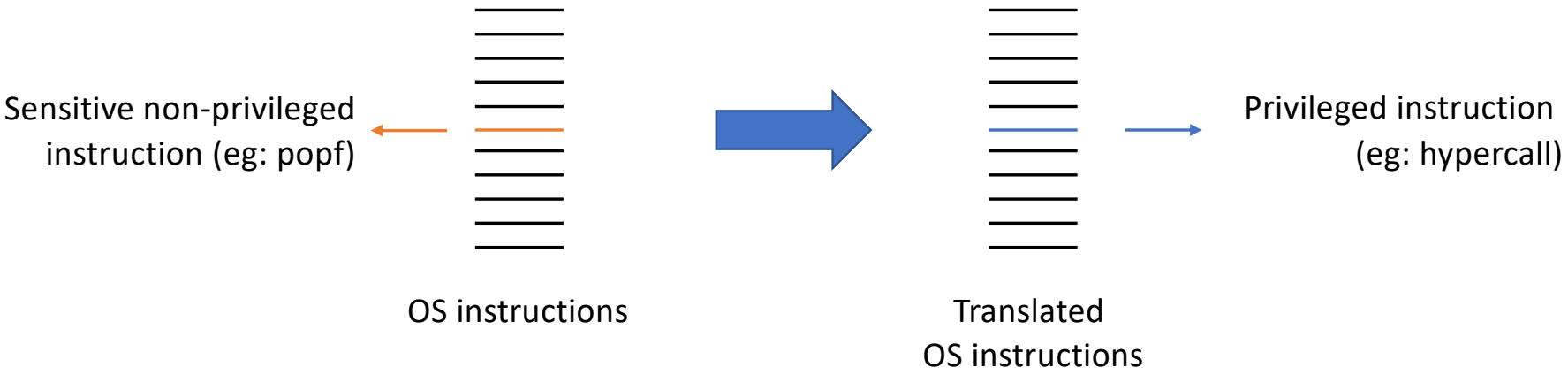
Group	Instructions
Access to interrupt flag	`pushf, popf, iret`
Visibility into segment descriptors	`lar, verr, verw, lsl`
Segment manipulation instructions	`pop <seg>, push <seg>, mov <seg>`
Read-only access to privileged state	`sgdt, sldt, sidt, smsw`
Interrupt and gate instructions	`fcall, longjump,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

Robin et.al. USENIX Security, 2000

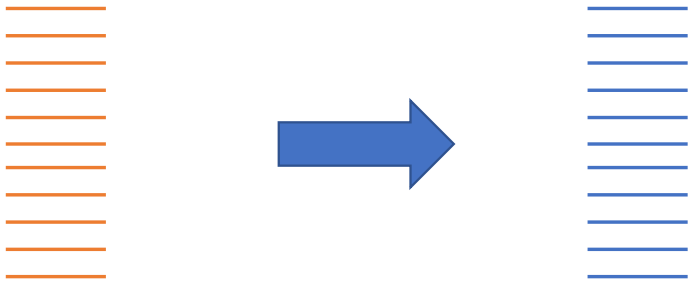
CPU paravirtualization in Xen

- Guest OS code modified to not invoke any sensitive + unprivileged instruction
  - Any privileged operation traps to Xen in ring 0
- Hypercalls: guest OS voluntarily invokes Xen to perform sensitive ops
  - Much like system calls from user process to kernel
  - Synchronous: guest pauses while Xen services the hypercall
- *Paravirtualizaition: Breaks transparency*
  - *Need to separately maintain XenoLinux*
  - *Cannot run Windows*

Problem setup: Binary translation



Similar problem as in architecture emulators, language runtimes



Java byte-code instructions /
Another architecture's instructions

Target architecture
instructions

Binary translation (QEMU)

```
addi r1,r1,-16    # r1 = r1 - 16
```

PowerPC instruction

(1) Parse into micro-codes

```
movl_T0_r1        # T0 = r1
addl_T0_im -16    # T0 = T0 - 16
movl_r1_T0        # r1 = T0
```

microcode instruction

(2) Append hand-written C codes, compile for x86

```
void op_movl_T0_r1(void) {
        T0 = env->regs[1];
}
```

Hand-written C code for each microcode

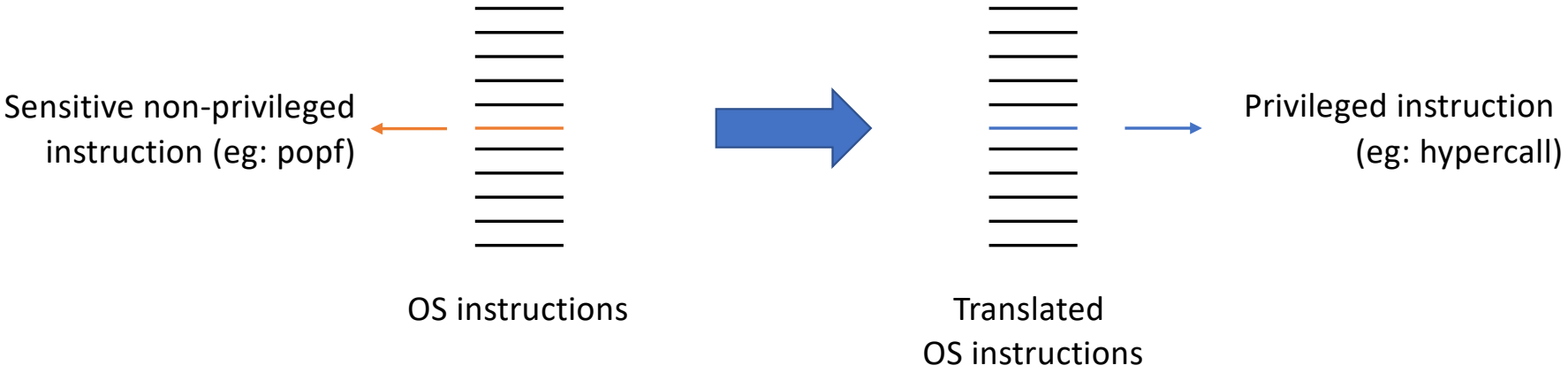
```
# movl_T0_r1
# ebx = env->regs[1]
mov 0x4(%ebp),%ebx

# addl_T0_im -16
# ebx=ebx-16
add $0xfffffff0, %ebx

# movl_r1_T0
# env->regs[1] = ebx
mov %ebx,0x4(%ebp)
```

(3) Optimize

Our problem is simpler



Same architecture

- No need to translate most instructions
- No need for doing optimizations

Static binary translation can’t handle all cases

- Code generated at runtime / dynamically loaded code
  - `sudo modprobe hypnos # loading a kernel module`
- Self-modifying code: Windows patch service
- CPU state may not be known statically:
  - 64-bit (long mode) / 32-bit / 16-bit (compatibility modes). Instructions are decoded differently in each mode: size of operands, etc.
  - `cli` : 8 different outcomes based on the state of CPU, and in particular based on the values of %cr0.pe, %cpl, %eflags.iopl and %eflags.v8086*
- Dynamic binary translation: Translate the binary as it executes!

* Bringing Virtualization to the x86 Architecture with the Original VMware Workstation

Dynamic Binary Translation-- Example

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

C program

```
89 f9 be 02 00 00 00 39 ce 7d ...
```

Binary representation

```
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
```

Assembly instructions

Dynamic Binary Translation– Example (2)

```
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
```

Assembly instructions

isPrime(49)
Translator

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

Compiled Code Fragment

Instruction cache friendly

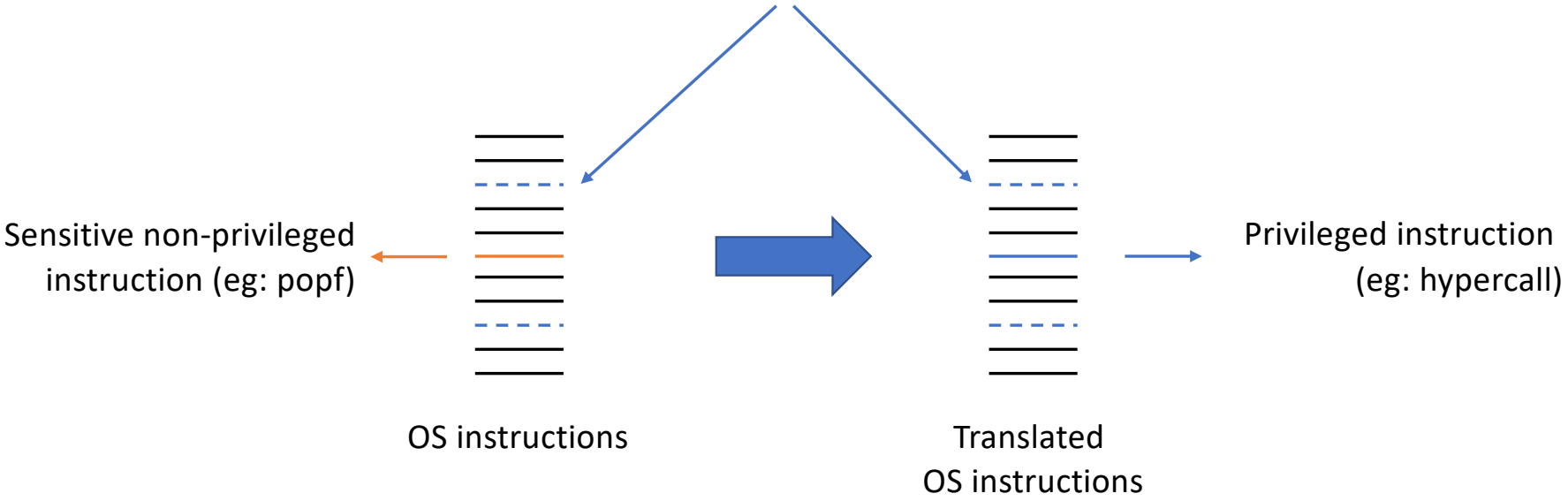
```
isPrime':  *mov   %ecx, %edi   ; IDENT
            mov   %esi, $2
            cmp   %esi, %ecx
            jge   [takenAddr]  ; JCC
                               ; fall-thru into next CCF
nexti':    *mov   %eax, %ecx   ; IDENT
            cdq
            idiv  %esi
            test  %edx, %edx
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

Comparisons with static binary translation

- Code that never executes never gets translated
  - Example: basic block “prime” in previous example
- DBT only needs to be done once for each basic block: CCFs are cached into a “translation cache”
- CPU state is precisely known at each instruction
  - 64-bit Windows XP Professional boot/halt translates 229,347 64-bit translation units (TUs), 23,909 32-bit TUs, and 6,680 16-bit TUs *

* A Comparison of Software and Hardware Techniques for x86 Virtualization

Optimizing “weakly exiting” instructions



Loads and stores trap many times

- Modifying page tables in shadow page table
- Doing memory mapped IO

Adaptive binary translation

- Note instructions that trap frequently
- Simulate them.
- Example: Modifying multiple PTEs => simulate as switch to ring 0, modify PTEs + shadow PTEs, switch back to ring 1

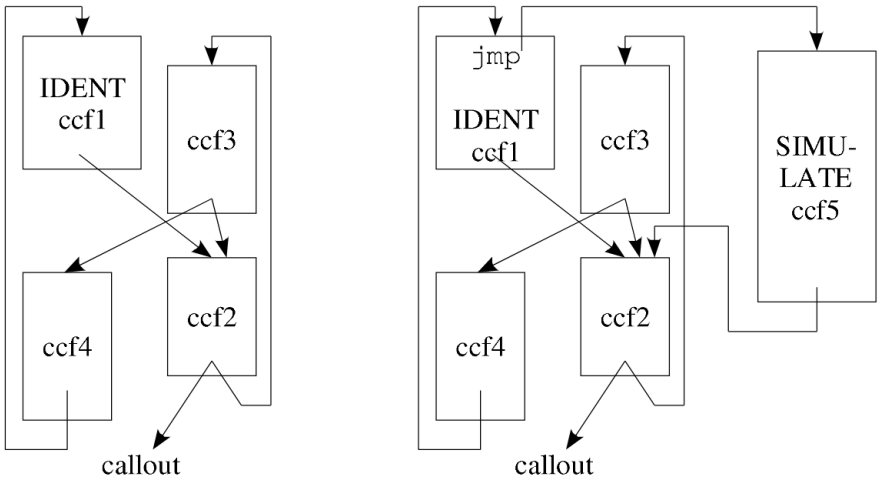


Figure 1. Adaptation from IDENT to SIMULATE.

Comparison of software and hardware techniques for x86 virtualization

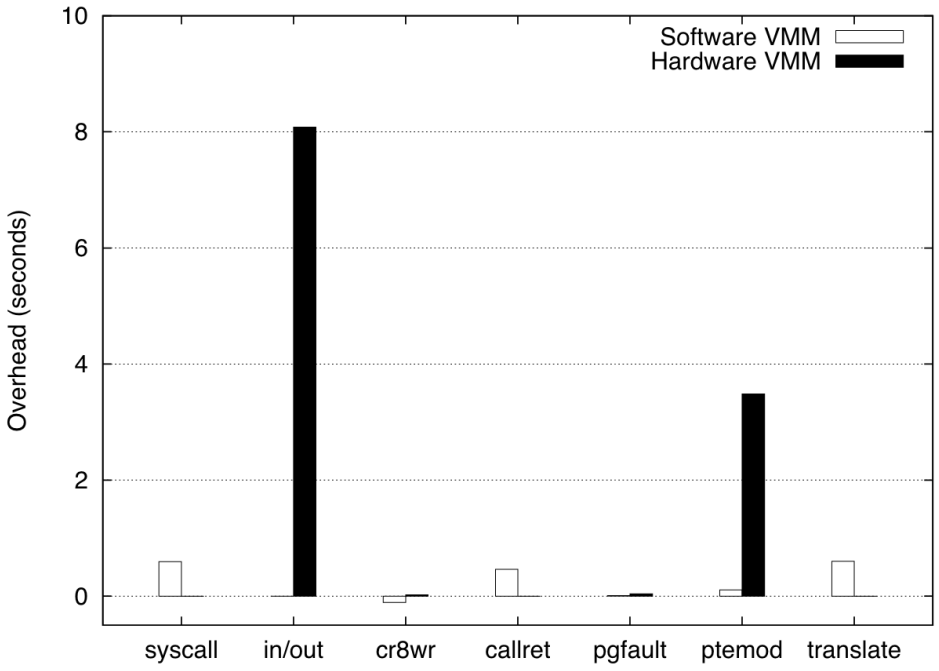


Figure 5. Sources of virtualization overhead in an XP boot/halt.

Hardware losses:

- VMEXITS >> TRAPS (in/out)
- Shadow page tables: exit at each ptemod (software can avoid some traps using adaptive DBT)

Software losses:

- Translate overhead
- syscall needs forwarding
- Call/ret: look up where to jump to from a hash table.

Relevance in hardware-assisted virtualization

32-bit x86 OS updating 64-bit PTE in shadow page tables: 12% faster for compiling kernel

```
* MOV 4(%ecx),%esi ; write top half of PTE
* MOV (%ecx),%ebx ; write bottom half
```

16-bit BIOS code

```
* OUT %eax,%dx
* OUT $0xed,%al
MOV %dx,$0xcfc
MOV %al,%cl
AND %al,$0x3
ADD %dl,%al
XCHG %ecx,%eax
XCHG %ah,%al
* IN %al,%dx
XCHG %al,%ah
XOR %cl,%cl
JMP %bx
```

Nested virtualization: 34% speedup in handling exits

```
* VMREAD -0x22222a(%rip),%rbx ; RIP
* VMREAD -0x222209(%rip),%rcx ; RSP
* VMREAD %rdx,%rdx ; RFLAGS
* VMREAD -0x1f229b(%rip),%rbp ; Intr
blocking
* VMREAD %rdi,%rdi ; Exit reason
* VMREAD %rsi,%rsi ; IDT vectoring
* VMREAD %rbx,%rax ; CS rights
ADD %eax,$0x2
* VMREAD %rbp,%rax ; SS rights
TEST %edi,%edi
JNZ 0x24
MOV %eax,$0x4404
* VMREAD %rax,%rax ; Interrupt info
```

Other measurements*

IO virtualization

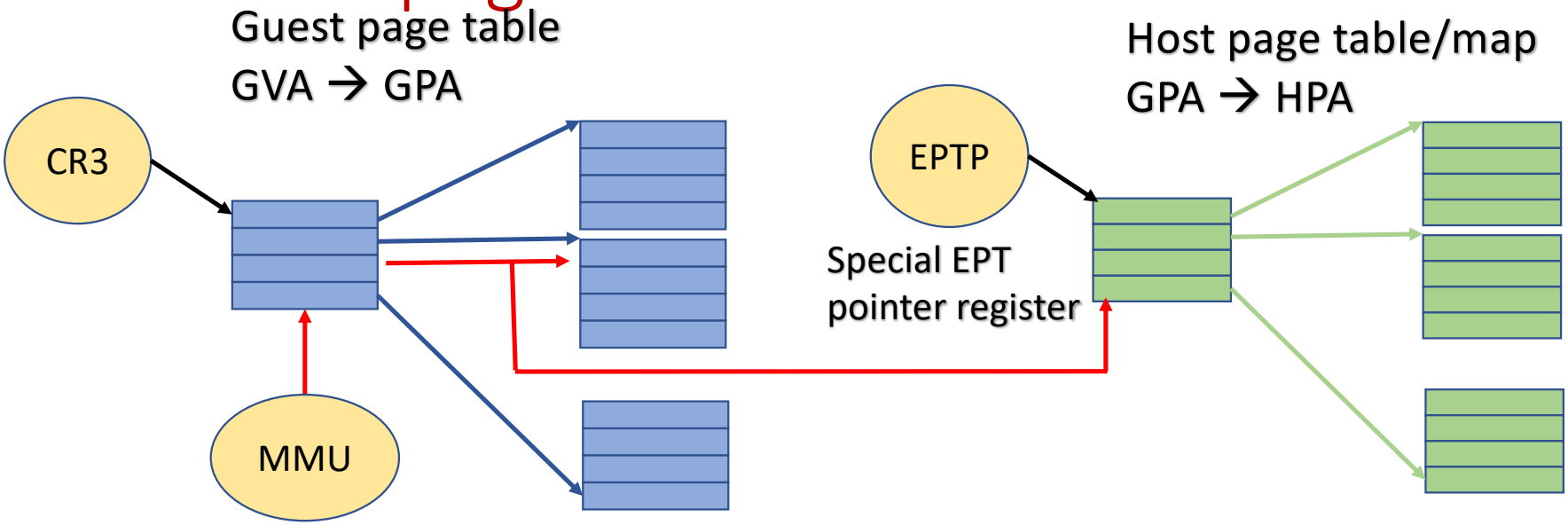
- e1000: 2.6 -> 1.97 exits per roundtrip
- vmxnet: 1.3 -> 1.2 exits per roundtrip

Complex: 10k LoC on top of x86 instruction decoder, translation caching, etc. already built by VMWare

Paravirtualization is much easier. Newer hypervisors like Firecracker do not do DBT.

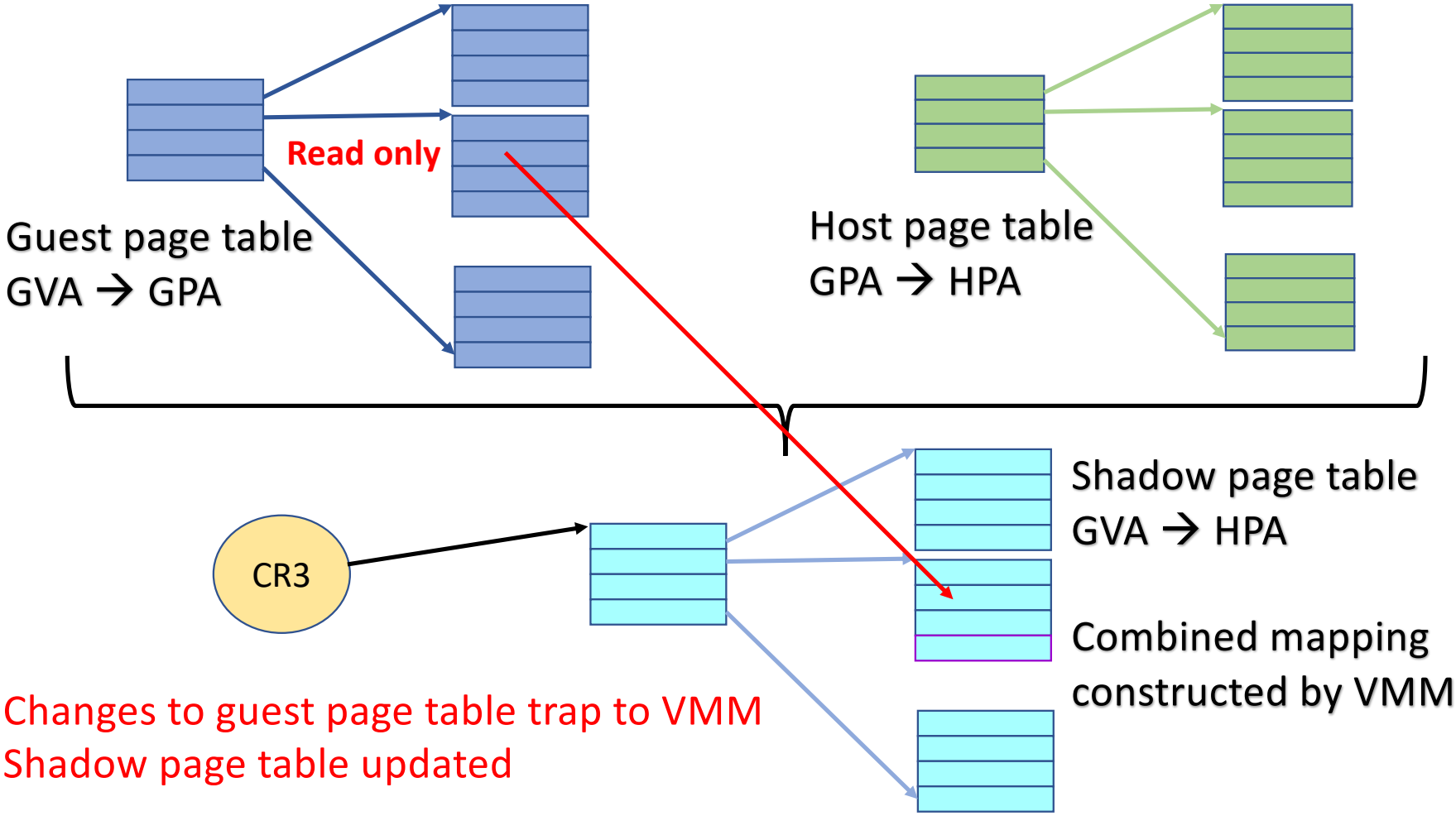
*Software Techniques for Avoiding Hardware Virtualization Exits

Hardware-assisted memory virtualization: Extended page tables



- Page table walk by MMU: Start walking guest page table using GVA
- Guest PTE (for every level page table walk) gives GPA (cannot use GPA to access memory)
- Use GPA, walk host page table to find HPA, then access memory page, then next level access
- Every step in the guest page table walk requires walking N-level host page table
- N-level page tables in guest/host result in page table walk of O(NXN) memory accesses

Memory virtualization without architecture support: Shadow page tables



Changes to guest page table trap to VMM
Shadow page table updated

Maintaining shadow page tables

- Guest writes to CR3, privileged operation traps to VMM
  - VMM marks the guest page table pages as read-only
  - VMM constructs shadow page table, sets CR3 to it
- Shadow page table can be built on demand
  - Start with empty page table, add entries on page faults
- Guest changes page table, traps to VMM, shadow entry updated
- Guest OS keeps multiple page tables of active processes in memory
  - On context switch, new page table used, but old page table still in memory
  - What about shadow page tables? How many in memory?
- Many design choices exist
  - VMM can discard old shadow page table on context switch, and rebuild it later (overhead during context switch)
  - VMM can maintain multiple shadow page tables of active processes (overhead to track changes to all page table pages)

Memory paravirtualization in Xen

- One copy of combined GVA→HPA page table maintained by guest OS
  - CR3 points to this page table
  - Like shadow page tables, but in guest memory, not in VMM
- Guest is given read-only access to guest "RAM" mappings (GPA→HPA)
  - Using this, guest can construct combined GVA→HPA mapping
- Guest page table is in guest memory, but validated by Xen
  - Guest marks its page table pages as read-only, cannot modify
  - When guest needs to update, it makes a hypercall to Xen to update page table
  - Xen validates updates (is guest accessing its slice of RAM?) and applies them
  - Batched updates for better performance

Xen: memory performance and scalability

- Fork/exec performance (Table 3, Xen)
  - Slower (198 µs) than Linux (143 µs) because the guest needs to trap for updating PTEs
  - Faster than VMWare (874 µs) because Xen can batch updates.

- Space-efficient
  - Shadow page tables maintain two copies for every page table
  - Xen: Only 20kB state per VM
  - Able to run 100s of (simple) VMs on a server with 2GB RAM

Fault-tolerant virtual machines

Faults are common.

Assume your laptop crashes after 5yrs ≈ 1800 days

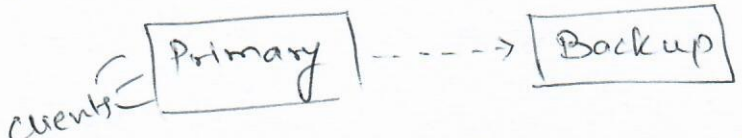
If you have 2000 similar laptops, you expect more than one to crash every day!

Types of failures

Fail stop: Just stops - detach power cable/network cable / fan broke → Focus of replication

Other faults: Software bugs
correlated: certain inputs crash the system

Replication approaches



1. State tranfer: Primary sends state (memory contents) as checkpoints

Single client op may change lot of state. ex: increment every element of a large array

2. Replicated State machine

Primary sequences and sends operations.

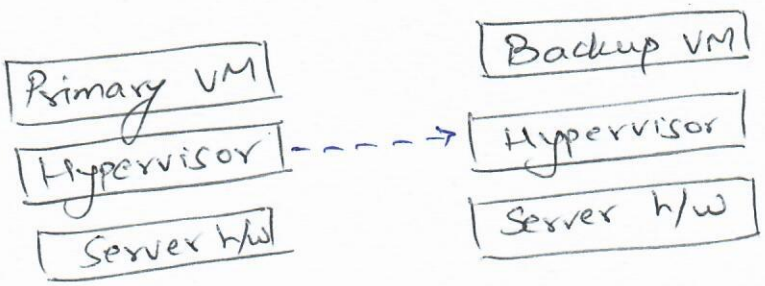
Same start state
→ same order of operations
→ each op is deterministic
⇒ same end state

Benefit: Operations are smaller than state

Tough to get right:
Execution shall never diverge.
⇒ Identify every source of non determinism

Application level state machine
- Database reads and writes
- Other non determinisms do not matter (like scheduling order of independent applications)
- Need to modify applications

Machine level state machine



- Completely transparently to the VM?
- No modifications. But more sources of non-determinism

Goal:

- Identical CPU state - register
  memory contents
  ⇒ execution never diverges

Basic idea: Send operations. Apply in same order on primary and backup.
No need to send anything for deterministic instruction like ADD

Sources of non determinism

- Interrupts,
- Client requests
- Internal (get time-of-day)
- Racy threads ⇒ order of load/store

Thread 1	Thread 2
① eax = mem[pos] (→ initially zero)	① eax = mem[pos]
② eax++	② eax++
③ mem[pos] = eax	③ mem[pos] = eax

① ② ③ ① ② ③ ⇒ mem[pos] = 2

① ① ② ③ ② ③ ⇒ mem[pos] = 1

⇒ Any memory access can be racy in a multi core system

⇒ Source of non determinism

Paper gives up on multicore.

Only Single core ~~Hardware~~ VMs to reduce ops

Divergence would be a disaster!

Student thought submission was ON TIME

TA think submission is late after fail over

Moodle VM

Submit assignment ⇒ 11:59 On time

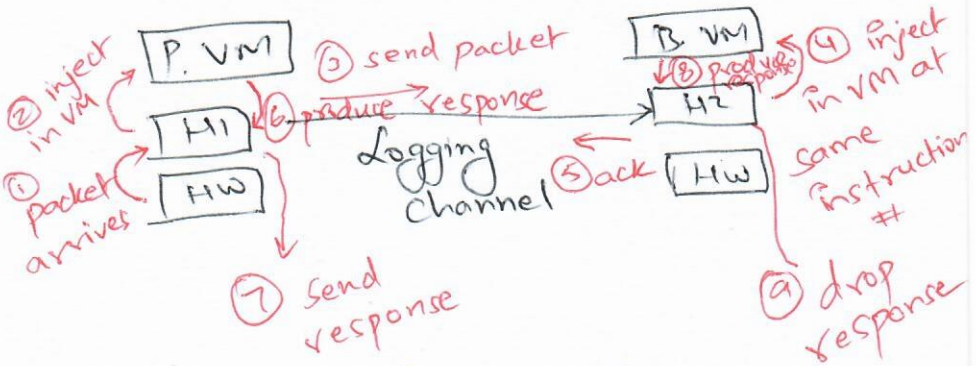
Submission closes 12:00 pm

Moodle VM

~~Submit ass~~

Submission closes 12:00 pm

Submit assignment ⇒ Late

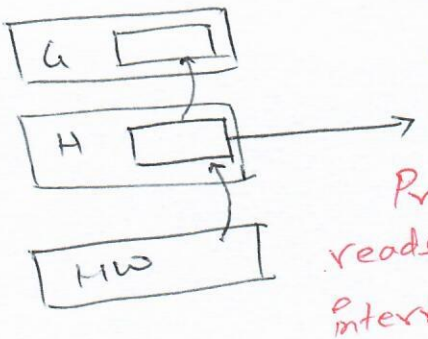


Backup replaying non deterministic events in same order.

Logging channel message

- Interrupt vector (n/w or timer...)
- Instruction #
- Data for Interrupt (n/w client packet)

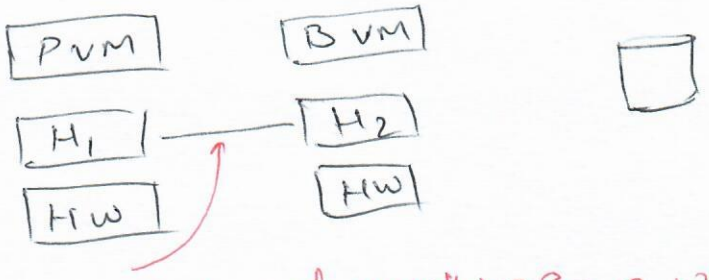
Bounce ~~before~~ buffer



Hw shouldn't write directly (DMA) in guest memory

Problem if guest reads memory BEFORE interrupt.

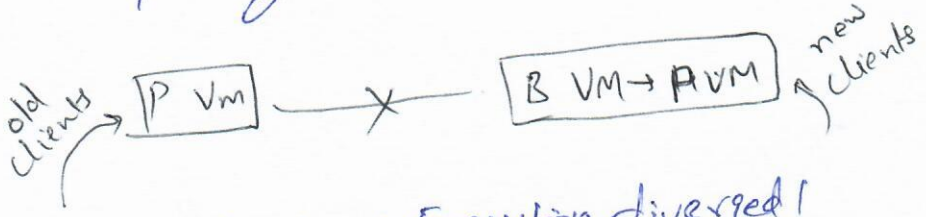
Fail over



Have not heard anything in a while
timer interrupts should happen regularly

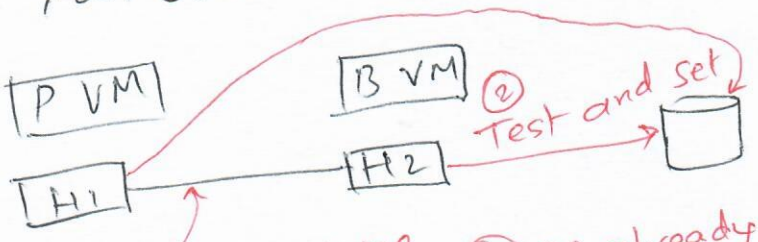
Problem for backup

Is primary down? Or is n/w down?



Split brain → Execution diverged!

Fail over



① not heard anything in a while

③ If already set, kill self

④ Become primary

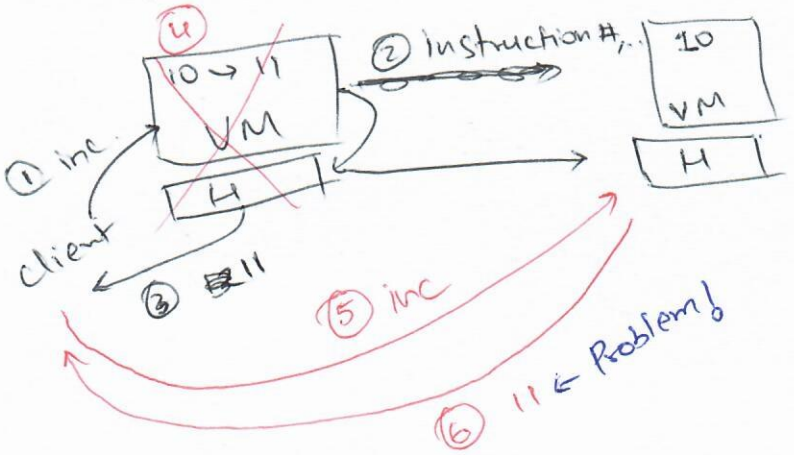
Can backup ack before injecting in VM? ⑤ before ④?

- Yes. Backup will play eventually.

Can primary send output before hearing back ack? ⑦ before ⑤?

- No.

Split brain example



OUTPUT RULE

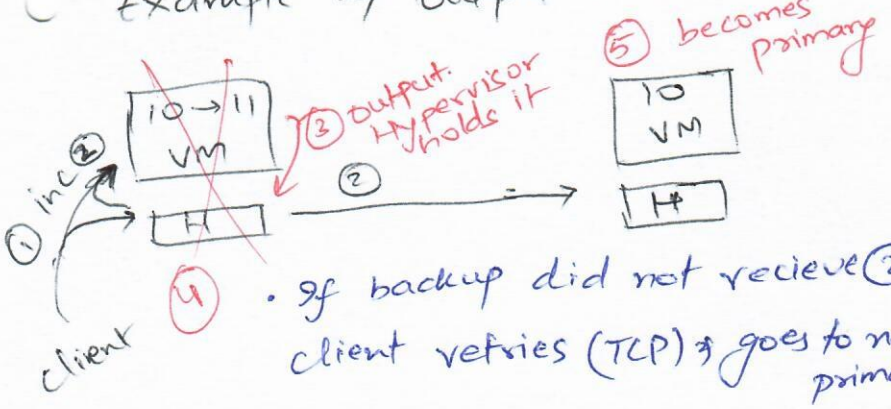
Primary cannot output UNTIL recieving ack from backup for all previous log entries

→ Synchronous replication: constraints of performance.

→ Applications play with output rule

- maybe don't wait before replying to read-only operation
  Eg: Zookeeper
- Hypervisor has no idea of semantics
  ⇒ waits

Example w/ output rule.



- If backup did not recieve (2) client retries (TCP) ⇒ goes to new primary
- Backup recieved (2) ⇒ 2 cases

Backup recieved (2), generated output

- After becoming primary ⇒ send to client
- Before becoming primary → Hypervisor trashes it

Guest TCP doesn't recieve ACK ⇒ retries after becoming primary ⇒ send to client

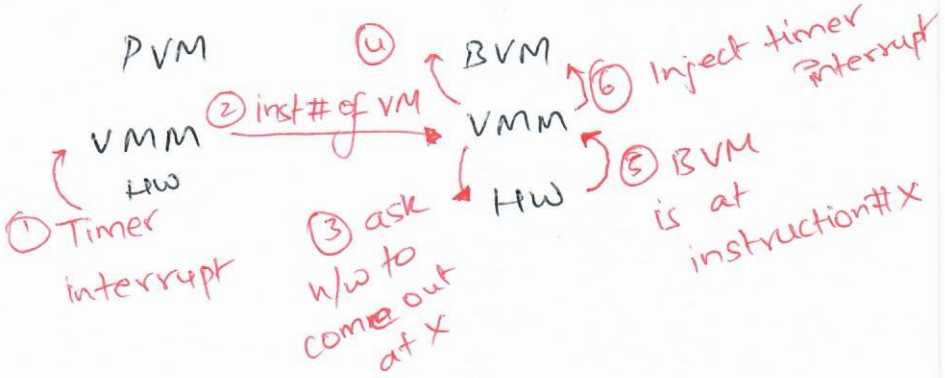
- Does backup have to wait?

yes.

Can only execute iff there is atleast 1 pending log entry

VMM programs CPU to jump out after instruction #X is completed by guest ← next log entry

- Example



Bank

Problem: From PoV of server, S_1,
n/w partition ≡ fault
Has S_2 failed?
Allow withdrawal?

↳ what if both thought other has failed
- Double withdrawals overdrawn account

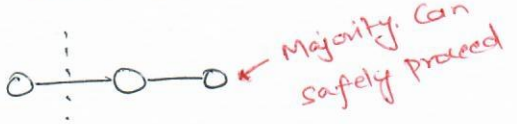
Cannot make PROGRESS ∴

→ Thought insurmountable for a long time
- Need external intervention
- Not auto FT

BIG Idea

Use majority vote

2f+1 workers - Resilient to f faults



Can never have 2 majorities

Sequential Consistency. (2)

* Don't want to talk about internals of storage
* Judge by behavior seen by clients.

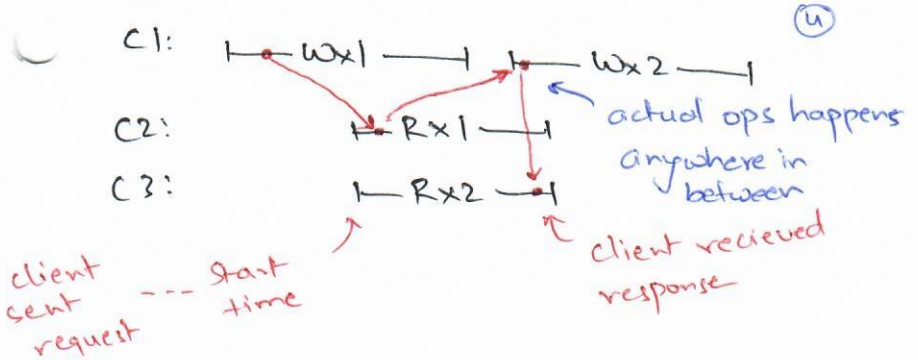
Create an illusion of single node storage

→ System is SC if all execution histories are Linearizable. (3)

- One can find a total order of all ops
  * matches real-time for non-overlapping ops
  * read sees last write

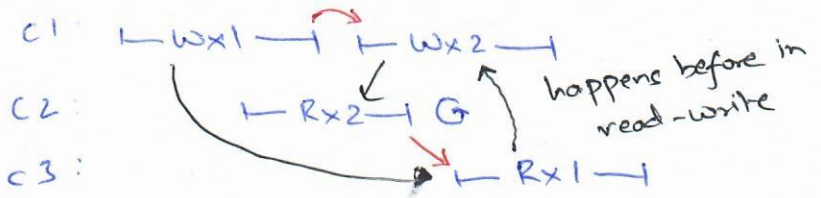
---

(4)

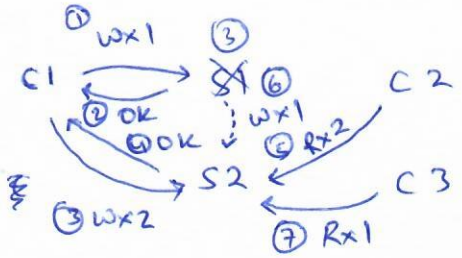


⇒ Linearizable

happens before in real time (4)

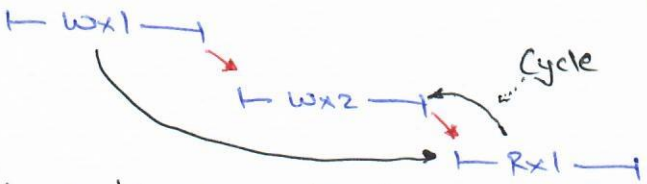


Cycle ⇒ NOT linearizable



- S1 responded to C1 before replicating at S2
- Replication message Wx1 reaches after Wx2, Rx2 to S2

(4)

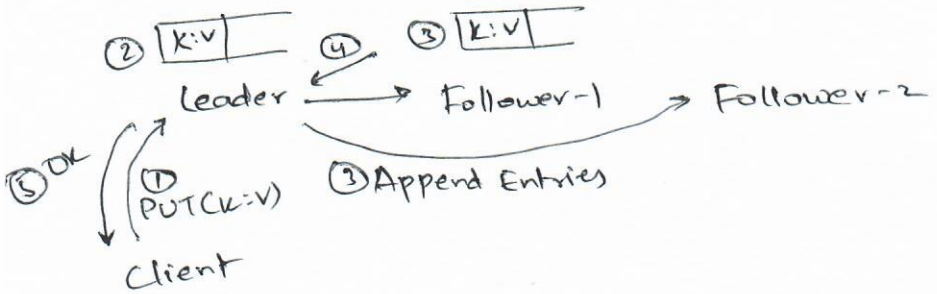


⇒ Stale reads are not allowed

- This can happen in Dynamo. Wx1, Wx2 Consumed by different partitions.
- Will not happen in Raft.

Minority partition Wx2 will not succeed

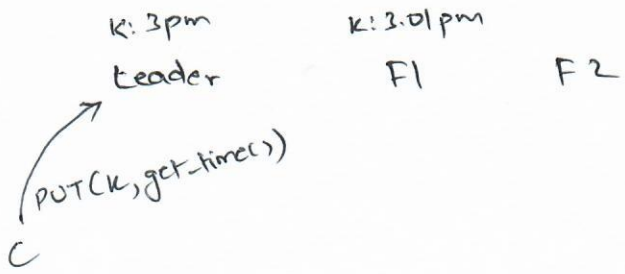
Raft architecture: Replicated log ③



---

④

Log is a sequence of operation on a deterministic state machine.

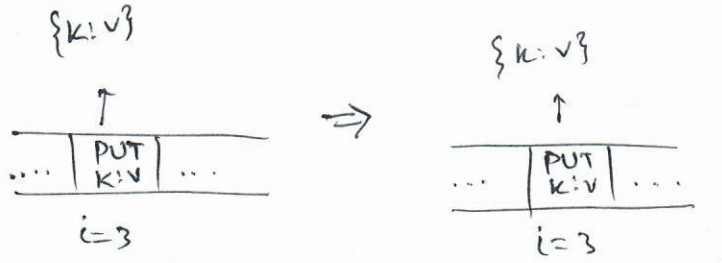


---

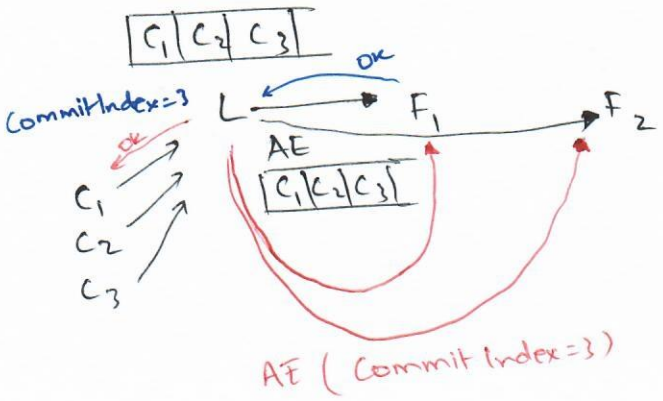
③

State machine safety: If a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index.

State machine safety



Normal operation



What if F2 is partitioned / dead?

L maintains nextIndex

F_1 : nextIndex = 3

F_2 : " = 0

L Keeps attempting AppendEntries to F.

Progress until majority

What if F2 was down for a long time? (1)

Send snapshot + log.

snapshot: state

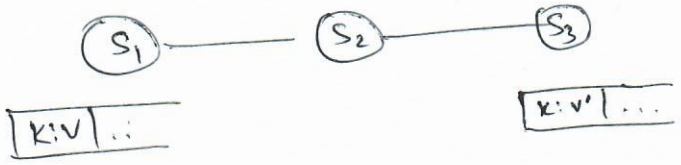
log: ops after snapshot

What if leader crashes? (2)

- Elect a new leader

Election safety: at most ONE leader can be elected (for a given term)

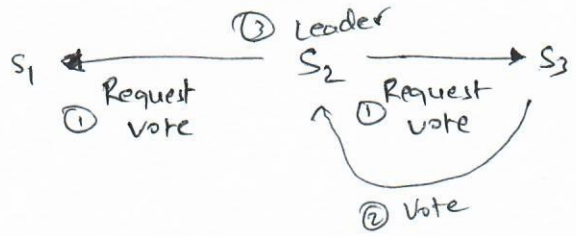
Both S_1 and S_3 think they're leaders (1)



Accepted conflicting values! BAD

How to detect a crashed leader? (2)

- Haven't recieved heartbeat



Wait for election timeout. Increment term. Start an election

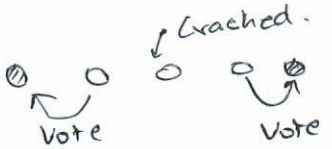
How to ensure single leader in a given term? (3)

Two majorities will have common node

Each server only votes once in a term.

Election Safety

→ What if there is a tie? (3)



→ Another election timeout.

Another election for a new term

Can we tie forever? Randomized timeouts

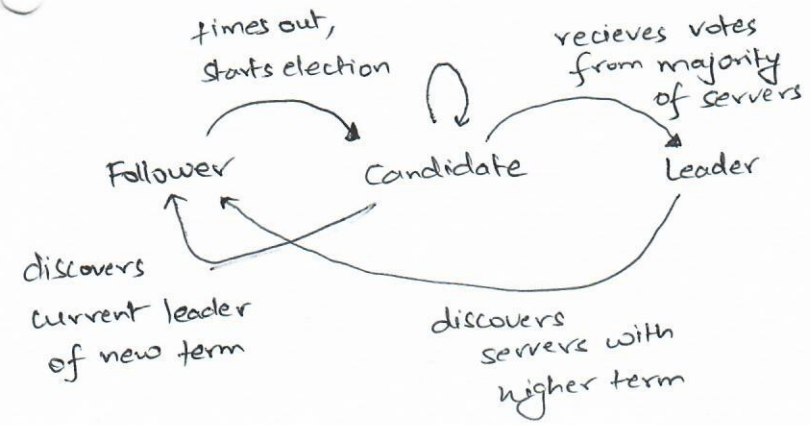
What is a good election timeout? (2)

1 ms? Timeout all the time.
No progress.

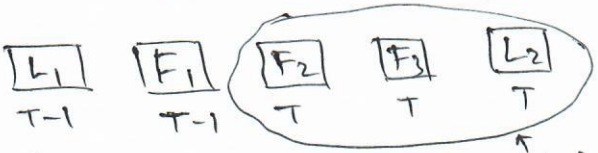
1 month? Long time without a leader.
No progress.

150 ms - 300 ms. ✓ Good enough range to avoid simultaneous timeout.

(2)

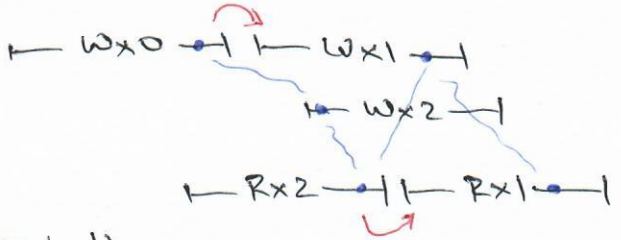


There CAN be 2 leaders in diff. terms. Why is this not a problem? (3)



Cannot commit
Cannot build a majority

Voted for L2.
Have "seen" term T.
Will reject AE from L1



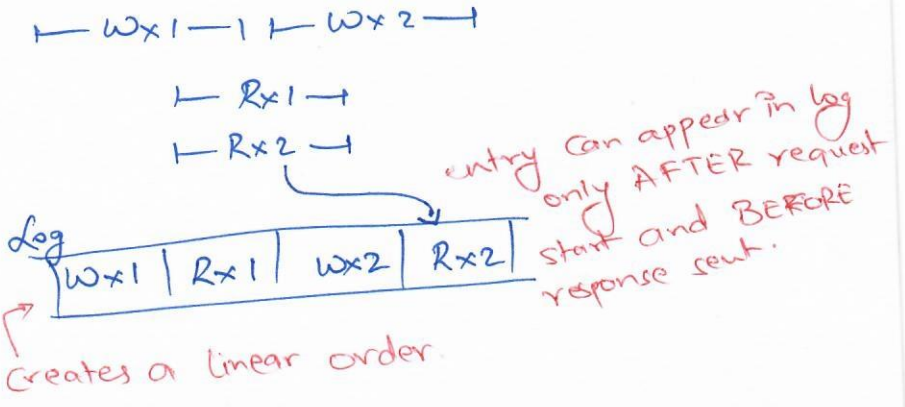
⇒ Linearizable.

In concurrent writes, write that started later can be executed first.

In Raft;

- Wx1 went to a leader
- Leader crashes
- New leader elected
- Got Wx2 first
- Then Rx2
- Then Wx1 is retried with new leader

Log implies linearizability



State (Elections + log) (4)

nv log[] → Keep in disk. Might need to catch up others if I become leader

nv currentTerm → Must reject AE from old leader.

nv votedFor → Can't vote twice

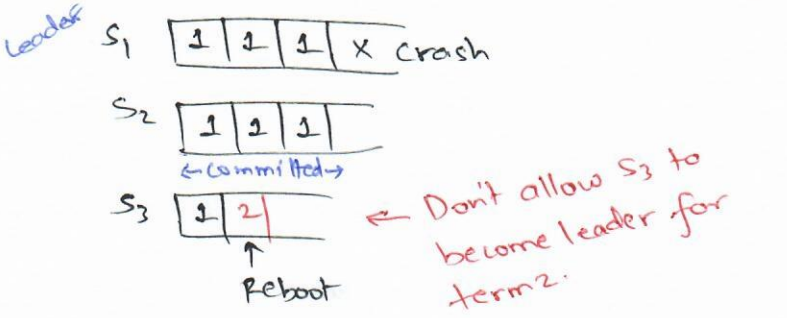
(3)

- Review state machine safety property.

- Election safety is good but not sufficient.

- Can I miss committed entry during leader replacement?

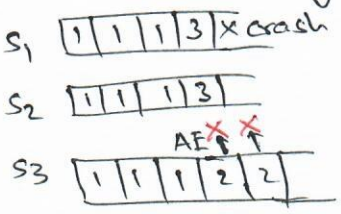
(3)



S_2 will not vote for S_3 since S_2 is more caught up.

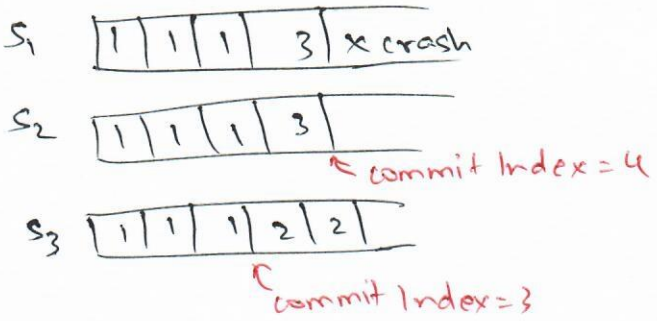
Who is most caught up? (4)

Idea 1: One with longest log?



For term 4,
Cannot pick S_3
· No knowledge of committed entry in term 3

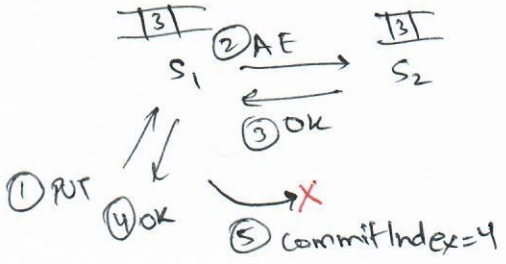
Idea 2: One with higher commitIndex? (3)



Only vote if Candidate commitIndex ≥ my commitIndex

Idea2: One with higher commitIndex (4)

Problem: S_1 may have crashed BEFORE it could tell S_2 about commitIndex = 4



Why not ack clients only after advancing commitIndex of majority? (4)

- Slower
- 2 quorums (2 Round trips)
- If someone in earlier majority crashes, I'll have to create another majority

Idea B (Raft). (3)

I'm more caught up if for my last log entry

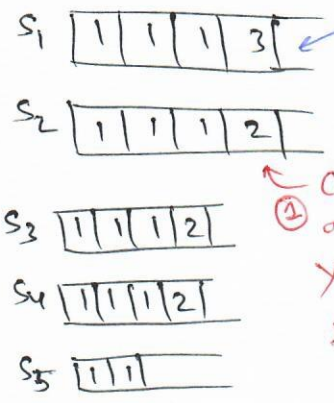
- my term num > their term num ✓
- if equal term num,
  my log index > their log index ✓

(2)

Candidate send (last log index, term no. of last log entry)

in RequestVote RPC

Why is this correct? ④



② Then this can't happen.

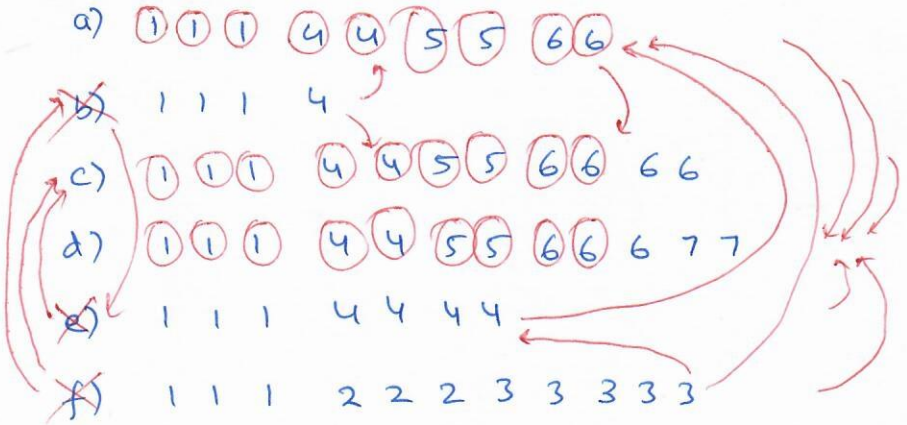
Only S_2, S_3, S_4 can become a leader. for term 3 and they'll force their logs.

① Can this be a committed entry?

Yes ⇒ ② majority had this entry Before voting for term 3.

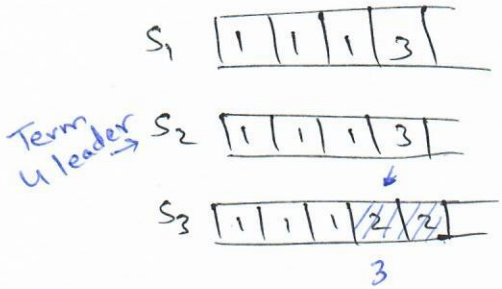
Otherwise, they'd reject AE(2)

ALL ~~EE~~ POSSIBLE LEADERS have committed all entries



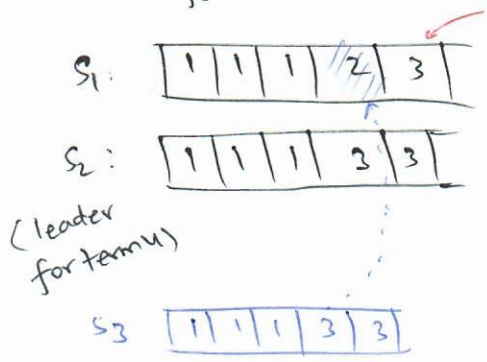
Leader completeness property!

What to do about diverging logs ③

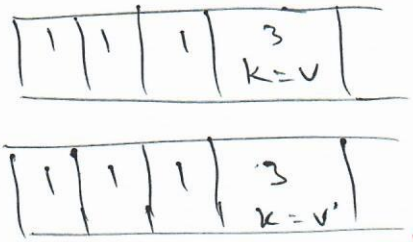


Append Entry is only applied if prevLogIndex and prevLogTerm match. Keep going backwards until they match.

Why is only checking last log index sufficient? (4)



Can this happen? (2)



No. Leader decides unique operation for <term, log index>

→ Safety properties (5)

- State machine safety.
  - Leader append only. Never rollback/delete committed entries
  - Election safety. Single leader in a term.
  - Log matching. Leader forces its log. Overwrites uncommitted entries
  - Leader completeness. Elect "caught up" server by checking <last log index, last log term>

What is scalability? ②

- I can support 1M users?
- I can run my program on 10k machines?

Similar to "what is a fast program"?

Let's try to define it analytically ②

Real hardware is messy

- cache access time << DRAM
- int ops << flops
- Frequency modulation

Analytically understanding becomes challenging

Algo guys: ③

- Build a simplified model of computation
- "Basic" operation take "constant" time
  +, <, x, ...
- Memory access is free
- Sorting ⇒ $O(N \log N)$

Parallel hardware is messier (3)

- Network topology
- Network condition
- Simultaneous execution

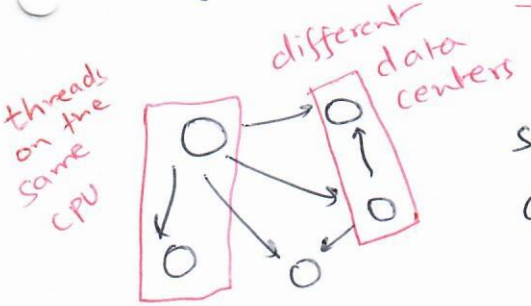
⇒ Load balancing, stragglers

Model of parallel computation (2)

- Basic operation take a constant time
- Primary reason for overhead is communication

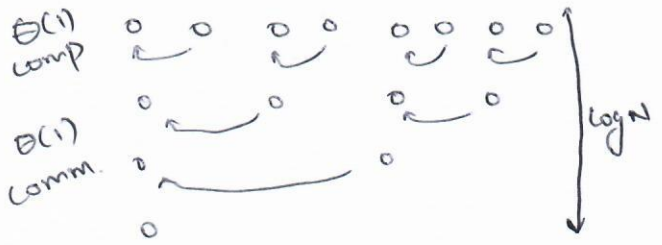
2) Sending 1 byte takes a constant time

Analytical model is over-simplification (2)



Still communication cost = t_c

Adding N numbers $T_s = \Theta(N)$ (3)



$T_p = \Theta(\log N)$

Speed up (S): $\Theta\left(\frac{N}{\log N}\right)$

Ideal speed up = Num. of processors (3)

$$\Theta\left(\frac{N}{\log N}\right) < N$$

Most processors are not "efficiently" utilized

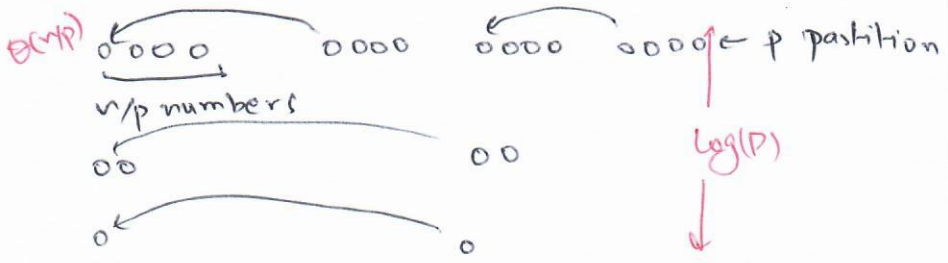
Efficiency, $E = S/p = \frac{T_s}{p T_p}$ (4)

Fraction of time processors are effectively utilized. $E = \Theta\left(\frac{1}{\log N}\right)$

For large N, $E \to 0$.

Most processors are idle.

Let's try with $p < N$ ②

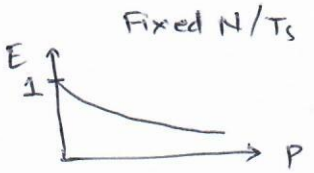


Computation = $\Theta(\frac{n}{p} + \log p)$

Communication = $\Theta(\log p)$

$$S = \frac{N}{\frac{N}{P} + \log p} = \frac{P}{1 + \frac{P \log p}{N}}$$ ④

$$\frac{S}{P} = E = \frac{1}{1 + \frac{P \log p}{N}}$$



All parallel systems

Scalability ③

- Rate at which problem size needs to grow, when we add a new processor. to maintain efficiency
- Lower the rate of growth, highly scalable.

For addition, $T_s = N = \Theta(p \log p)$

W

Also called isoefficiency function

Implicit assumptions ③

- **Shared memory**! Pass a pointer.
  Communication = O(1)
- **Uniform memory Access**
  - Place any task on any worker
- **No stragglers**
  - mismatch h/w, load, faults...

Reality check: Locality matters ③

CPU ↔ cache = 1 ns

CPU → DRAM = 100 ns

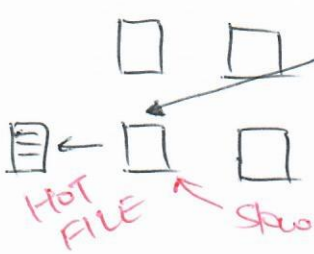
200M×

CPU → Data center net = 50 µs

CPU → Cross D/C WAN = 5-200 ms

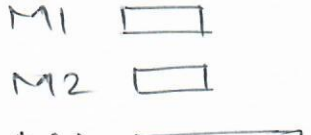
• Load balancing matters ②

→ 1000s of concurrent jobs on 1000s of machines



- Locality based scheduler tries to schedule every job on this node

Stragglers

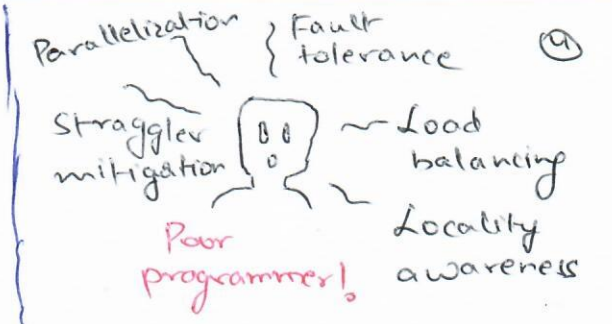


- Load imbalance
- Kernel bug
- Faulty disk/network
- Heating
- Slower CPU

+ Faulty

Can we build better abstraction?

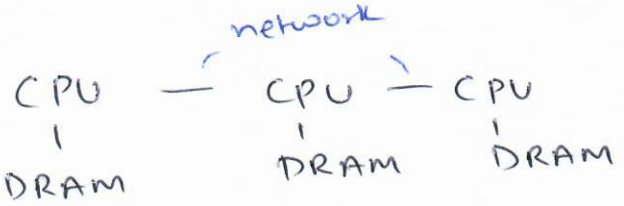
- Only think about business problem.
- Holy grail: Write a "serial" program that just works in cloud.



AB

1980s hot topic-

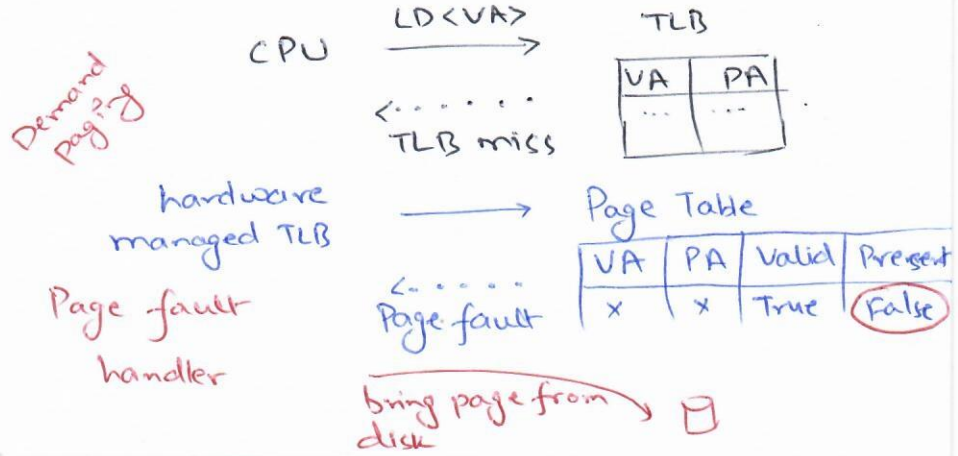
Distributed Shared memory (DSM)



Unified logical view of memory.

- Shared address space

One realization. Page-based DSM (4)



Page-based DSM (2)

Page-fault handler

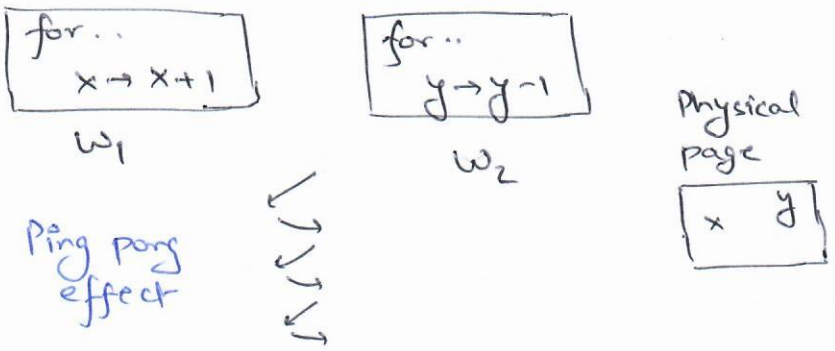
bring page from remote

- either interrupt remote CPU to service
- RDMA (more modern)

Benefits of DSM (5)

- Programmer is oblivious of distributed computation
- familiar programming model
- Program for local memory scales to DSM without modification
- Identical virtual space => Easy process migration

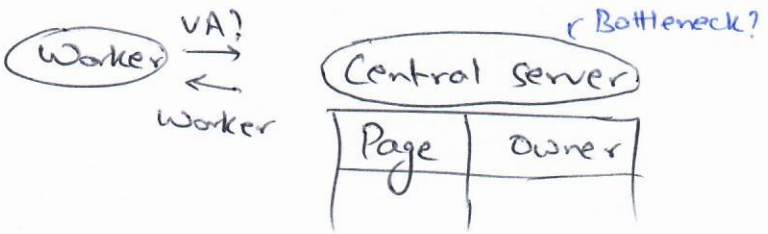
Problems - False sharing



Problems: Who has the page?

- Solve false sharing by smaller pages.
  - ⇒ Increased book keeping
- Static partition. hash(VA) → Worker
  - What if hash function is skewed?

• Free movement



Page is moved to writer.

Fault tolerance / straggler ②

Any failed (slow) worker
fails (slows) entire Job

Checkpoints for fault tolerance? ③

- How frequent?
- What about writes between two checkpoints?

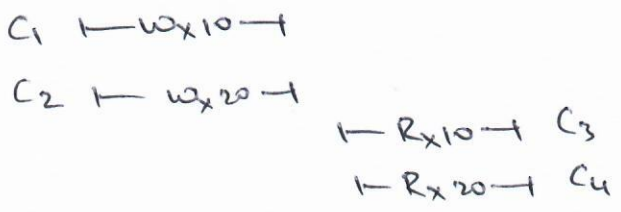
Full
- Checkpoints are expensive

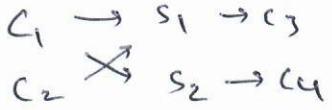
Replication for fault tolerance? ②

- Each page has three copies
- Each write needs to write thrice

⇒ Slow!

Consistency problems





Main Problems (DSM) ①

Concurrent writers and readers to same/adjacent VA

- False sharing
- Races
- Consistency problems

Single Writer. Immutable after that

Arbitrary fine-grained writes ①

- Hard to handle stragglers
- Recover from faults

- Regular expensive checkpoints?
- Replication overhead?

Only deterministic coarse grained updates

Spark: RDD ①

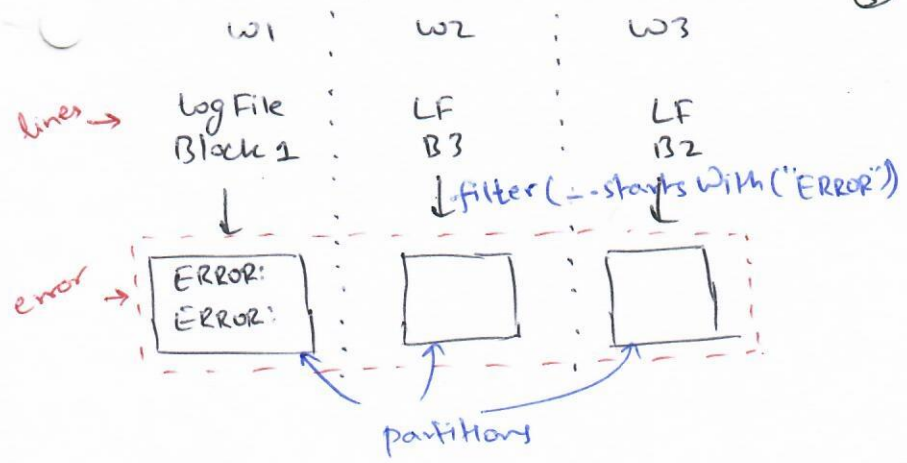
Resilient - fault tolerance

Distributed - Similar to DSM

Datasets - Collection objects

PULL UP HDFS error example

(3)



(2)

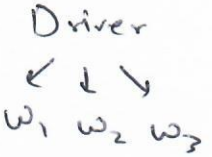
- Each partition is written locally by a single worker
  - no races
  - Good data locality
- Once RDD is written, it is immutable
  - no consistency problems

(3)

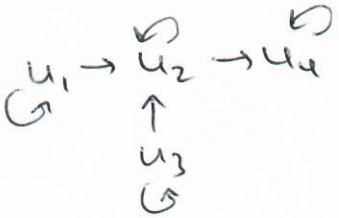
Manager-worker model (same as MR)

Driver

- manages worker
- keeps track of RDDs
  - what partitions)
  - partition function?
  - where is a partition?
    - Could be replicated
  - List of dependencies



Page Rank: measures popularity of pages



other pages point to it

- Expectations:
  - u_1, u_3 have equal page ranks
  - $u_2 > u_1, u_3$
  - $u_4 > u_1, u_3$

A Popular page points to it. → $u_4 > u_2$
It doesn't consider other pages popular

Page Rank Algorithm

- What is the probability that someone will visit the page?

Iterative:
- Start with equal ranks

Simulate a user at each step

$P = 0.85$ clicks a link at random

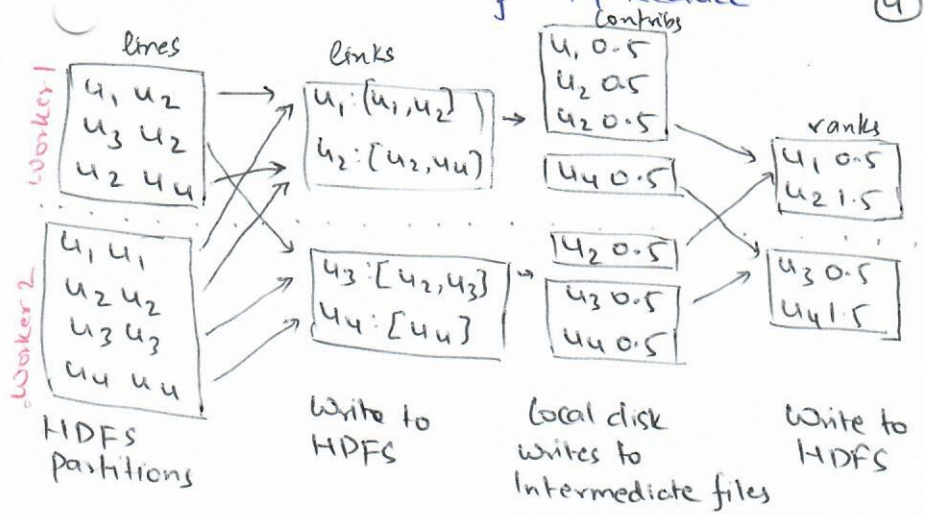
$P = 0.15$ goes to a random page

Algorithm runthrough

Links	$ranks_0$	$contribs_0$	$ranks_1$
u_1 [u_1, u_2]	u_1 1	u_1 0.5	$u_1 = 0.5 * 0.85 + 0.15$
u_2 [u_2, u_4]	u_2 1	u_2 0.5	$= 0.575$
u_3 [u_2, u_3]	u_3 1	u_2 0.5	$u_2 = 1.5 * 0.85 + 0.15$
u_4 [u_4]	u_4 1	u_4 0.5	$= 1.425$
		u_2 0.5	$u_3 = 0.575$
		u_3 0.5	$u_4 = 1.5 * 0.85 + 0.15$
		u_4 1	$= 1.425$

Iterative = Series of Map Reduce

(4)

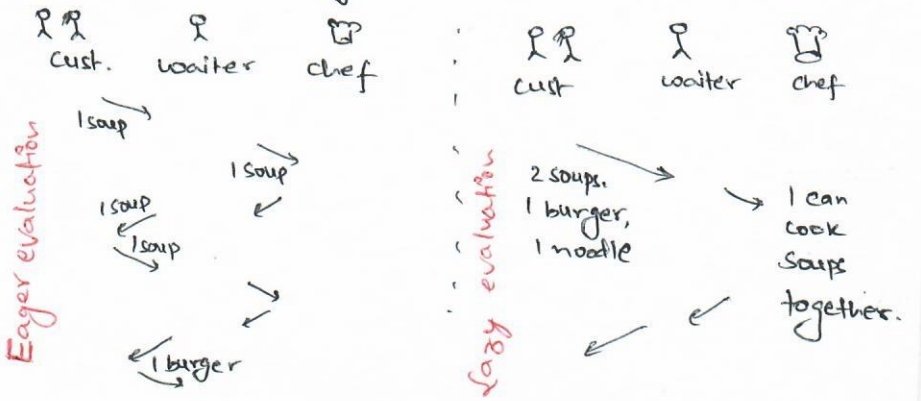


---

(3)

Before we look at demo. Remember

Spark does lazy evaluation



---

(15)

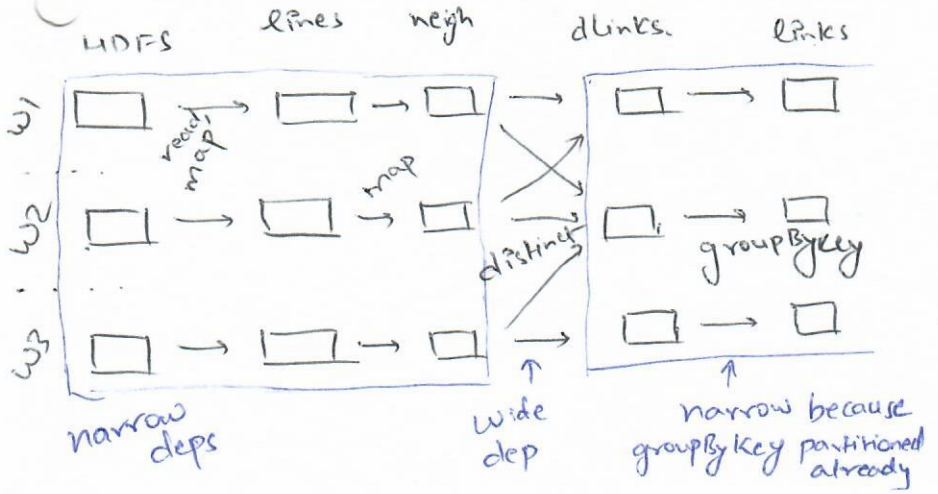
[redacted]

- [redacted]

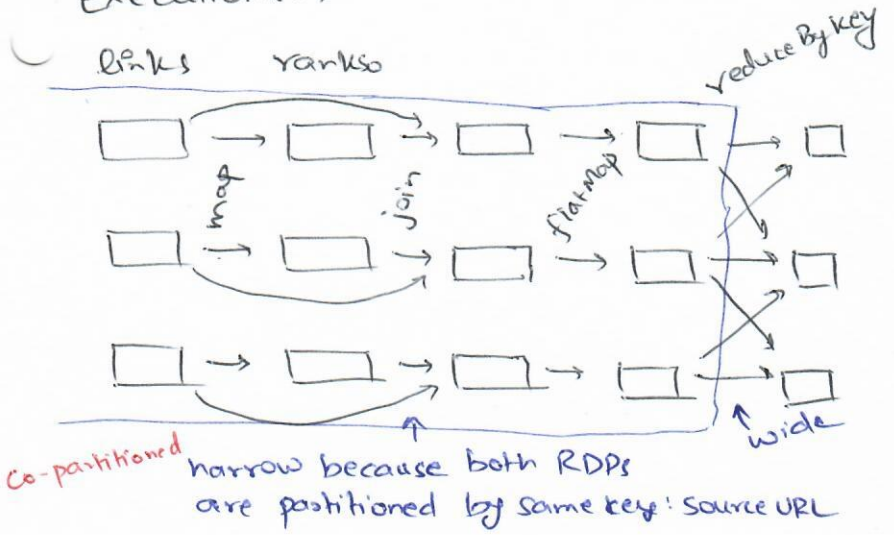
Transformations are just collected in lineage graph. Action executes.

PULL up page rank example

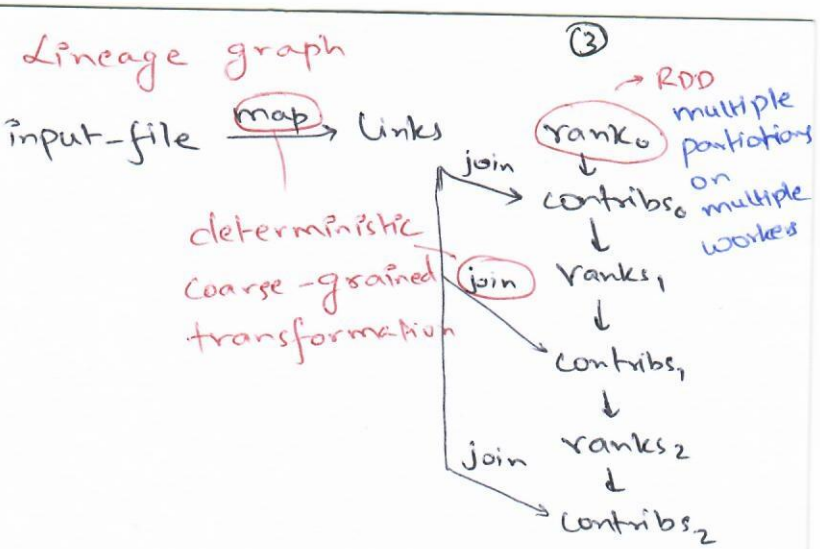
Execution (5)



Execution (?)



Lineage graph (3)



Lineage graph optimizations

(3)

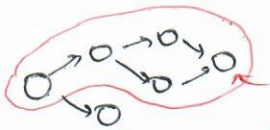
possible due to lazy eval.

→ map → map → ⇒ → map →

Tokenize sentence to words

Uppercase

Tokenize and uppercase



collect

skip evaluating this node (RDD)

Fault tolerance

(3)

- Recompute partition
- Ok because deterministic
- Long lineage chain

  ⇒ checkpoint occassionaly

ranks$_0$ $\Theta(N)$

↓

contribs$_0$ $\Omega(N^2)$

↓

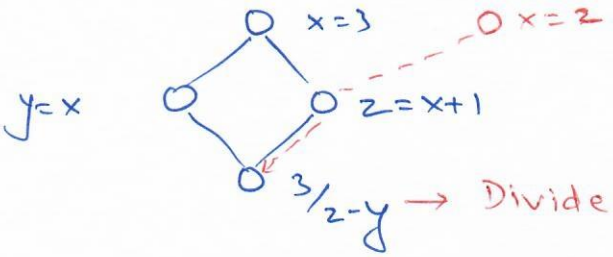
ranks$_1$ $\Theta(N)$

↓

contribs$_1$ $\Omega(N^2)$

cheaper than full checkpoint

Non-deterministic tasks/ immutable inputs



Making inputs immutable for every task tasks deterministic

Straggler mitigation ①

- Backup task
- Ok because w_1 and w_2 are writing the new partition locally
- Not modifying input (immutable)

DSM	RDD ②
Arbitrary fine-grained updates	Deterministic coarse-grained updates
Consistency issues from multiple writers/reader	Immutable. No issues
Expensive checkpoints Checkpoint all	Checkpoint "less" data in Lineage

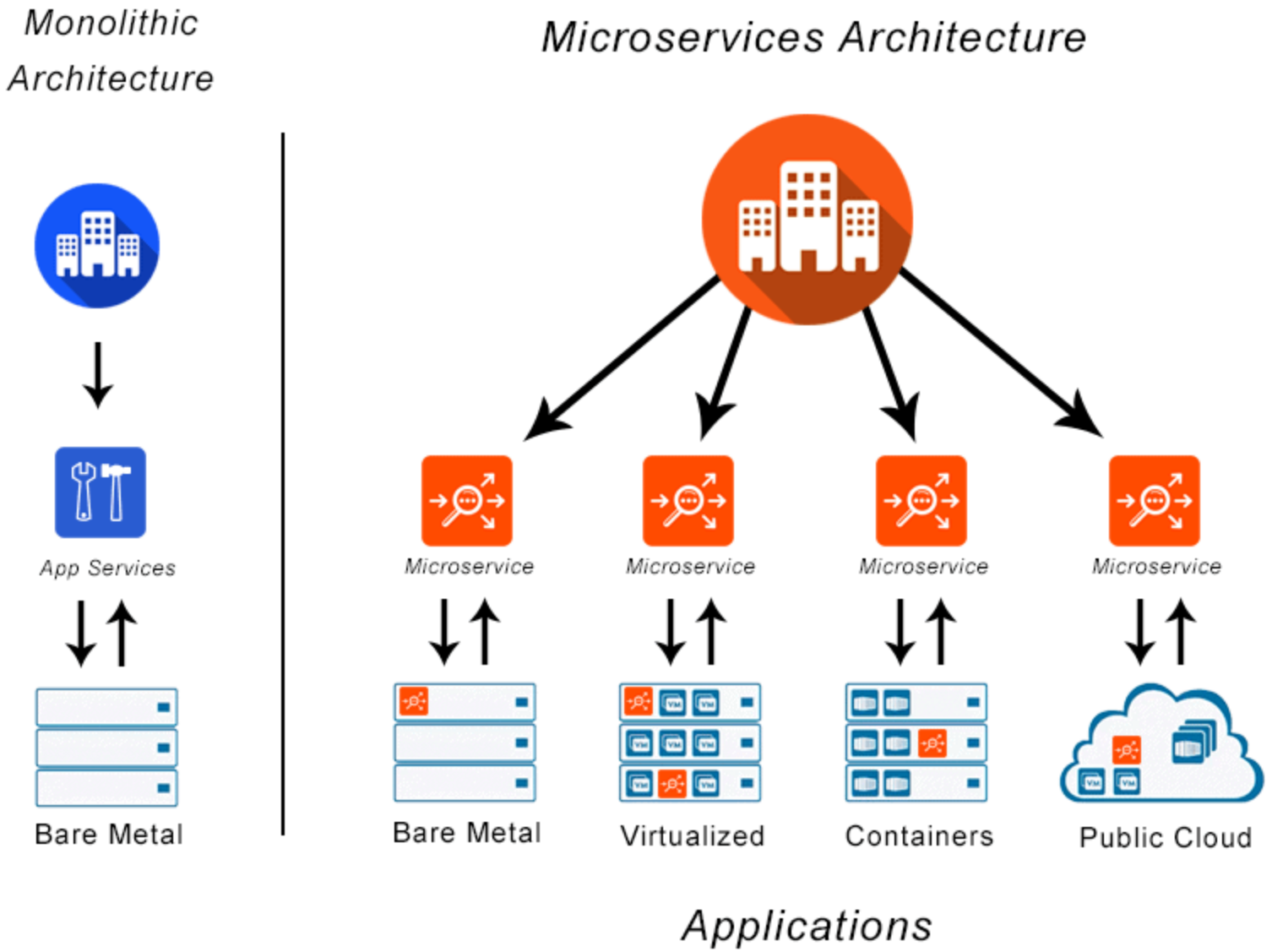
DSM	RDDs ②
Transparent Placement (may be unoptimal)	Partition control Locality aware
Replication for FT/ Stragglers	Re-exec / backup worker using Lineage

Unikernels

Variation in type of service

Service	Fault tolerance	Workload characteristics
Webserver	Stateless, ok to crash	CPU bound
ML inference service	Stateless, ok to crash	GPU bound
KV cache	Stateful. Ok to lose data across restarts!	Memory bound
Database	Stateful. Must not lose data across restarts!	Disk bound

Monolithic vs microservices



Microservices advantages

- Independently place
  - Place DB on servers with large SSD
  - Place ML inference on servers with strong GPUs
- Independently scale
  - User service vs tweet service

=> Each VM is only running one service to ease its placement / scalability etc.

Q: Identify microservices in the course project?

Problems with multi-application OS

OS provides high-level abstractions based on common case application behavior. *Common case abstractions can be wrong*:

- Assume temporal memory locality -> Evict pages to disk using LRU. Graph processing can access pages in random order. Same as hash tables.
- Assume temporal disk locality -> Evice file cache pages using LRU. `grep` is sequentially reading files. MRU disk block is better for evicting file caches.

OS is *conservative*:

- At context switch, OS saves all the processor state. If I know I am not doing any flops, I could have skipped saving floating point registers.

Exokernels

Applications can get better performance if they get direct access to the hardware.

Examples:

- Direct access to the disk instead of just to the file system.
- Direct access to the network packets instead of via TCP/IP stack

Optimizations:

- Which page to evict?
- What registers to save at context switch?
- Databases and webservers can skip file system's write-ahead logging

Exokernels

Primary challenge is to expose raw hardware without letting applications step on each other.

Many attempts but did not take off on desktops

Shaped many ideas

- Virtualization
  - Memory virtualization in Xen: directly expose page table to the guest
  - Memory ballooning in VMWare: guest decides which pages to evict (similar to upcalls in exokernel)
- Data Plane Development Kit: process packets in user space
- eBPF: Let apps push code into OS safely
- Unikernels

Unikernels

- Hypervisor already exposes raw hardware*. VM gets direct access to
  - CPU registers (VMX mode)
  - CR3 page table
  - IO devices through SRIOV
- Microservice architecture => single application in a VM.
- **Unikernels:** OS that supports only a single application.

Unikernel advantages

- Full control: Applications choose how to do paging, etc.
- Single protection level: System calls are as fast as function calls! (61 ns vs 1 ns)
- Smaller images: remove unused kernel modules
  - e.g., remove disk drivers + filesystem from key-value stores
  - => faster boot

Mirage [ASPLOS'13]

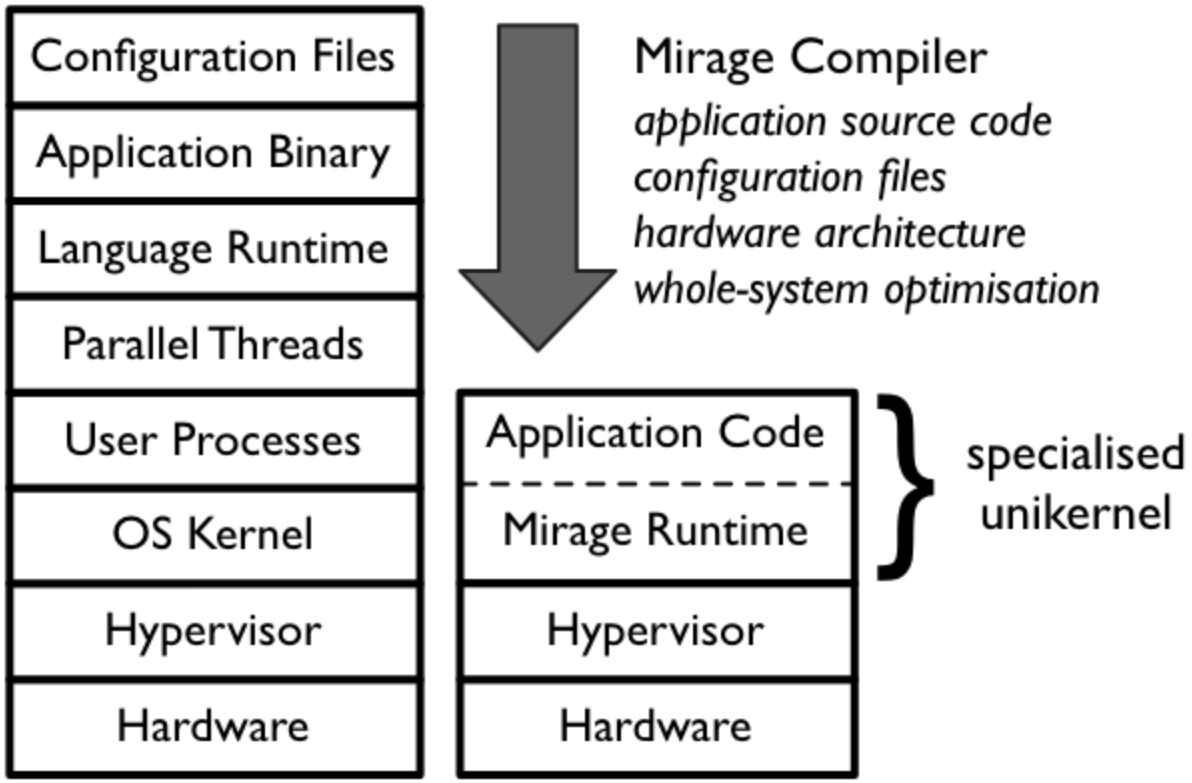


Figure 1: Contrasting software layers in existing VM appliances *vs.* unikernel's standalone kernel compilation approach.

Mirage

Dead-code elimination

```c
int foo(void)
 {
   int a = 24;
   int b = 25; /* Assignment to dead variable */
   int c;
   c = a * 4;
   return c;
   b = 24; /* Unreachable code */
   return 0;
 }
```

Re-write core OS in OCaml

Targets OCaml applications

Unikraft [Eurosys'21]

Goals

- Easy portability
  - Languages: C/C++, Go, Python, Ruby, Web Assembly, LuA
  - Applications: Nginx, Redis, SQLite
  - VMMs: KVM, Xen
- Easy customization
- Supports major applications and languages

Hard to isolate modules in Linux

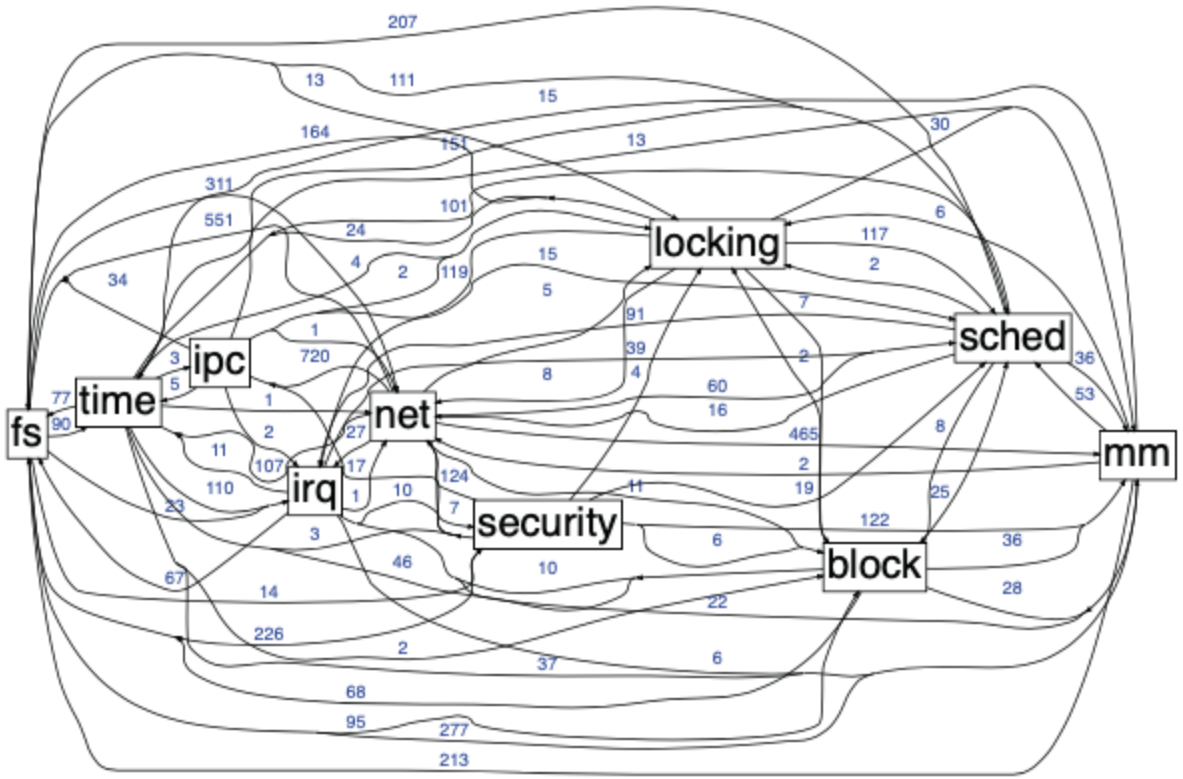


Figure 1. Linux kernel components have strong inter-dependencies, making it difficult to remove or replace them.

Importing one module imports everything else => Unable to eliminate most of the kernel

Unikraft: micro-libraries

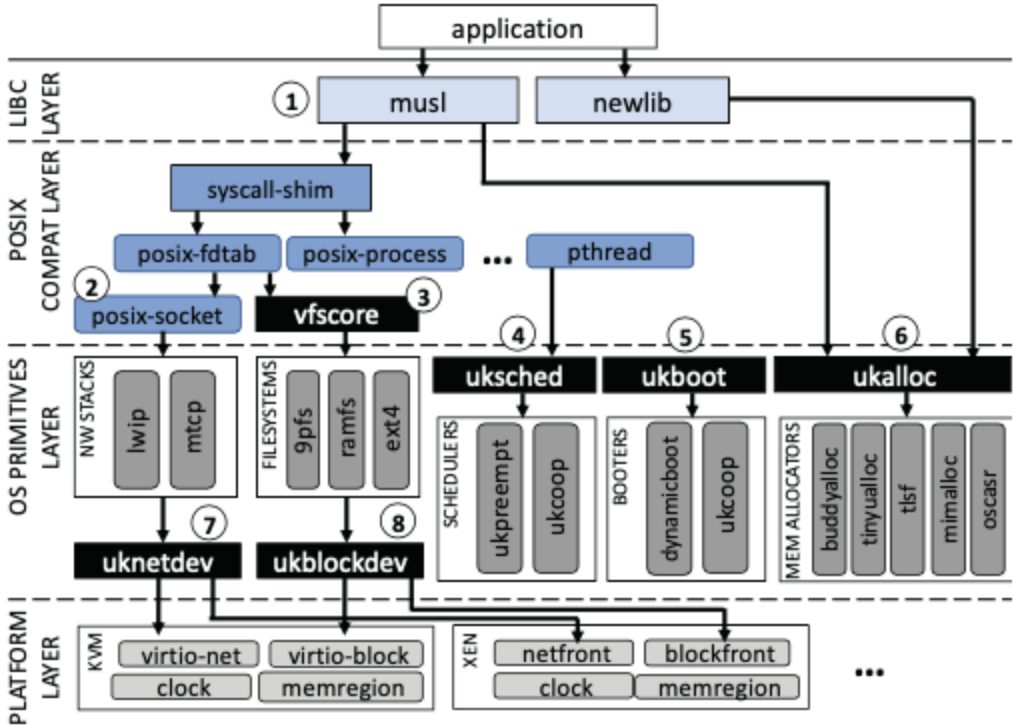


Figure 4. The Unikraft architecture (APIs in black boxes) enables specialization by allowing apps to plug into APIs at different levels and to choose from multiple API implementations.

Micro-libraries

Choices for libc

Many choices for OS primitives:

- Schedulers: preemptive / cooperative
- Memory allocators: buddy, tiny, etc
- File systems: ramfs, 9pfs, etc

Integrations with hypervisor: KVM / Xen

Unikraft dependency graph

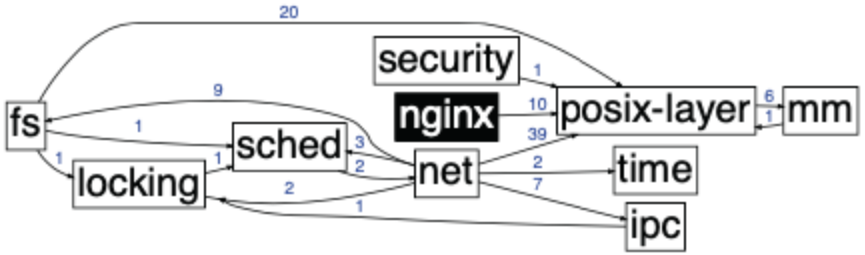


Figure 2. Nginx Unikraft dependency graph

Nginx does not need a disk => no block device!

Unikraft: Lower image size, faster boots

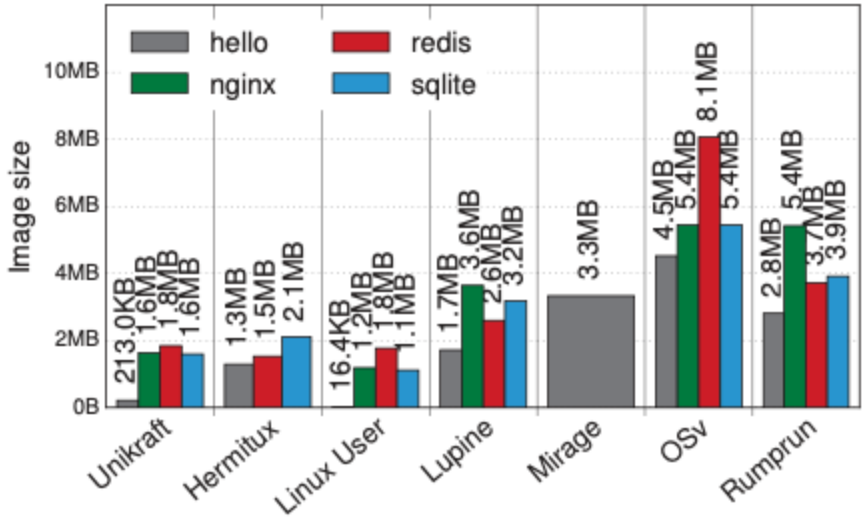


Figure 9. Image sizes for Unikraft and other OSes, stripped, w/o LTO and DCE.

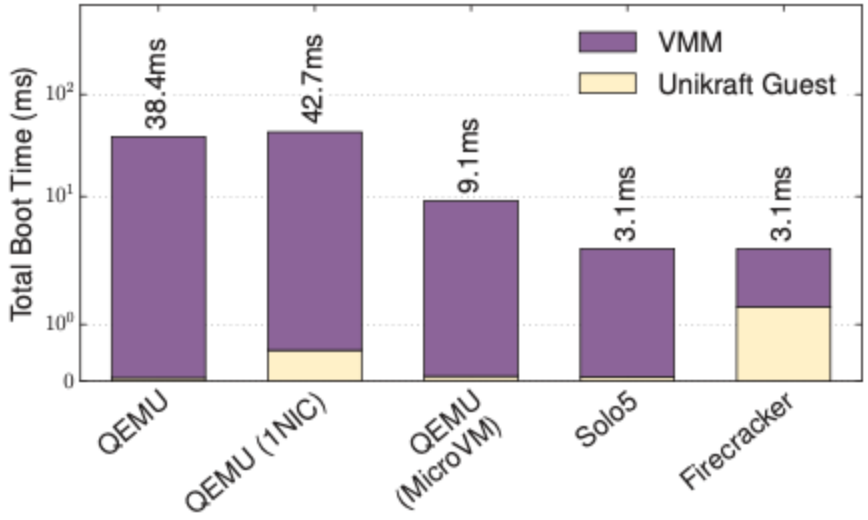


Figure 10. Boot time for Unikraft images with different virtual machine monitors.

- Linux takes around 330ms to boot on Firecracker.
- Higher VM density, just in time boot (for FaaS)

Specialization: boot time

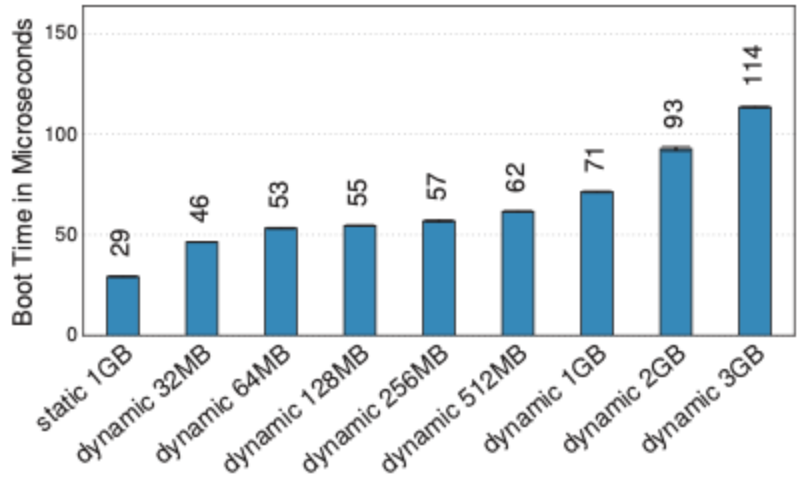
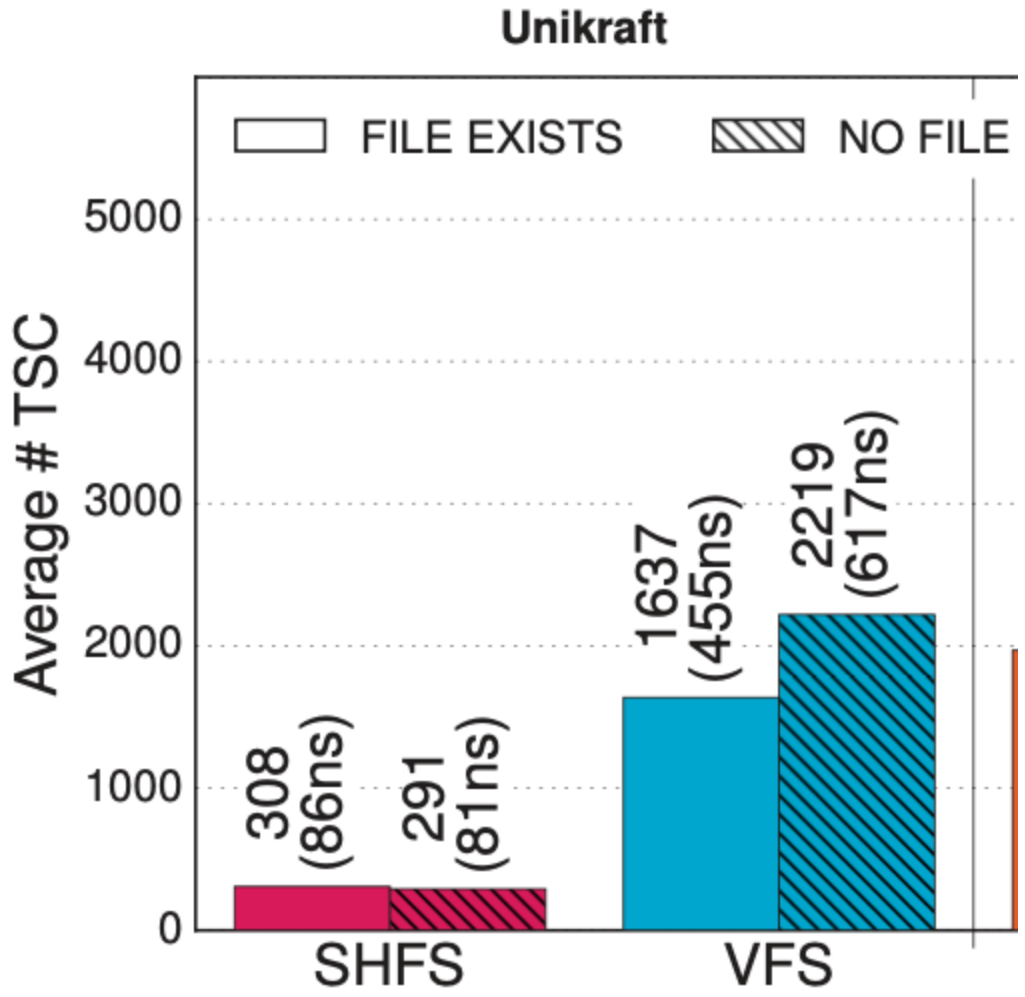


Figure 21. Unikraft boot times w/ static and dynamic page table initialization.

Static: VM image has the page table already built

Dynamic: Allow apps to change page table via `mmap`

Specialization: Specialized file system



Web cache: Does not require hierarhical file system. SHFS: just a hash table on blocks

Application throughput

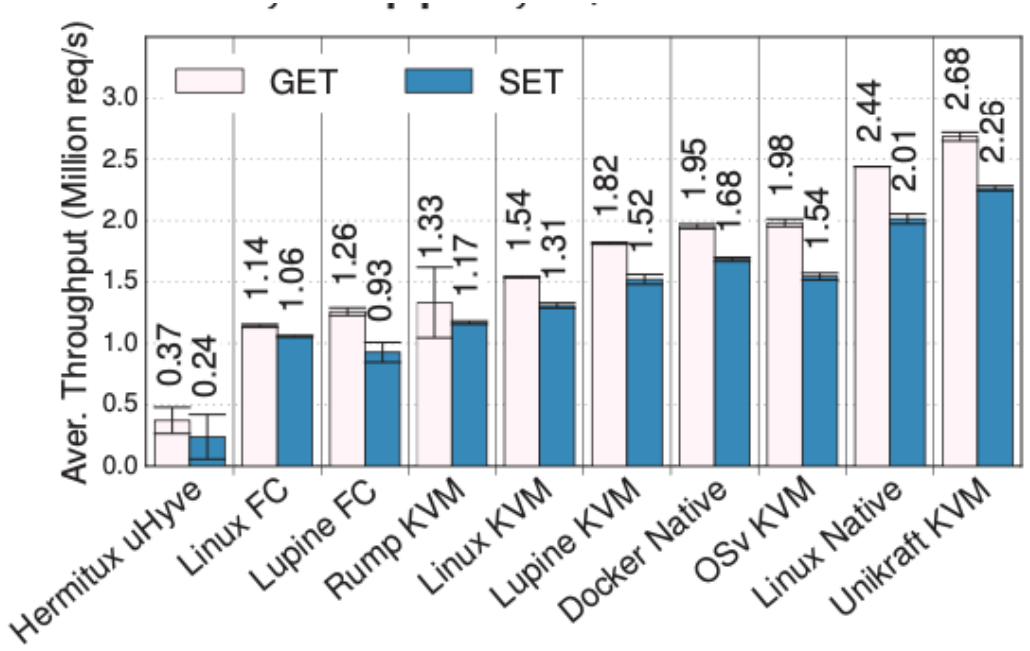


Figure 12. Redis perf (30 conns, 100k reqs, pipelining 16) with QEMU/KVM and Firecracker (FC).

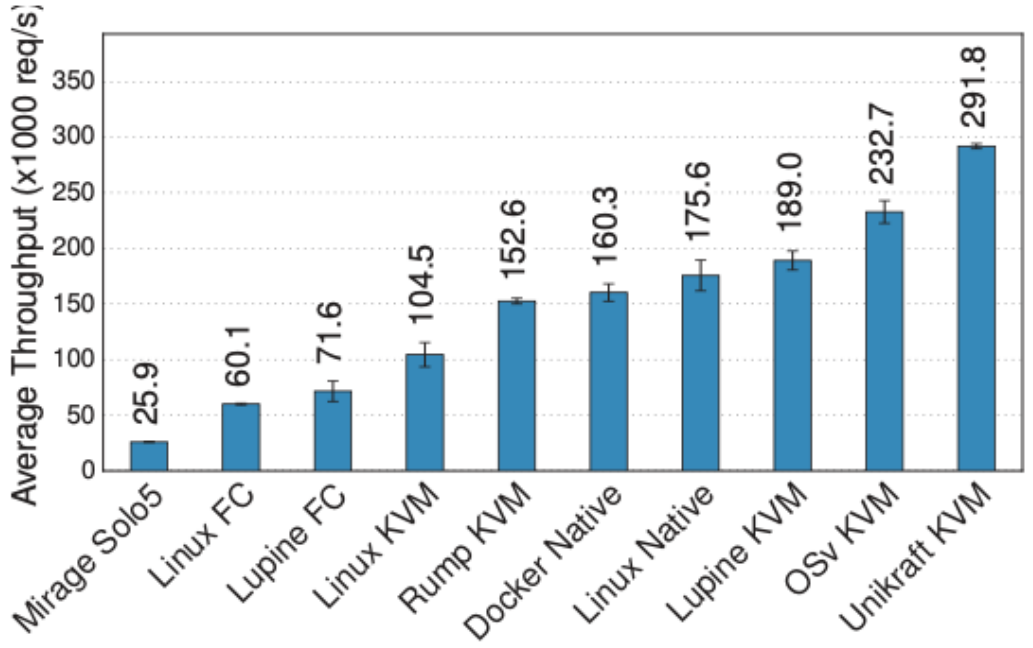


Figure 13. NGINX (and Mirage HTTP-reply) performance with `wrk` (1 minute, 14 threads, 30 conns, static 612B page).

- 30-80% faster than docker container, 70-170% faster than in a Linux VM
- Even faster than Linux native! *No syscall overhead*

What did we learn so far? ①

Batch computation

- Lots of stored data
- Jobs run for a long time

MR, Spark, Celery

Why streaming computation? ②

- Real-time decision making
  - Credit card transaction. Fraud?
- Live analytics
  - Trending topics on Twitter
  - How many web page visitors right now?
- Infrastructure monitoring
  - How many cars on this junction?

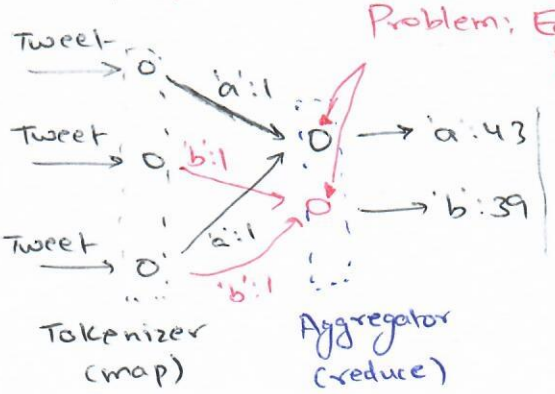
- Concerns ②

Freshness

- Time from data in to data out

- Fault tolerance
- Stragglers
- Scalability

Pipeline-model

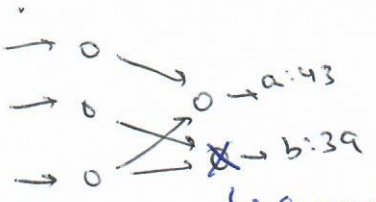


Problem: Each task has a "state". ③

"Continuous" operator model: New events update state

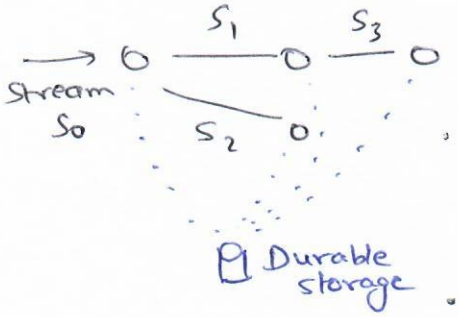
Fault tolerance / straggler mitigation ②

- Approach: Rerun tasks deterministic stateless



↳ Cannot rerun. Need state

Pipeline model (Producer-consumer) ③

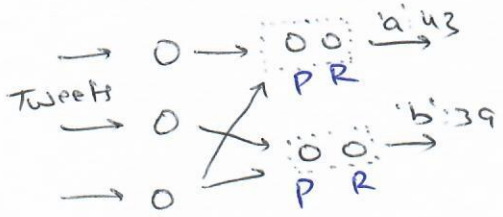


- Keep all storage in common storage
- Make workers stateless

✓ Real time

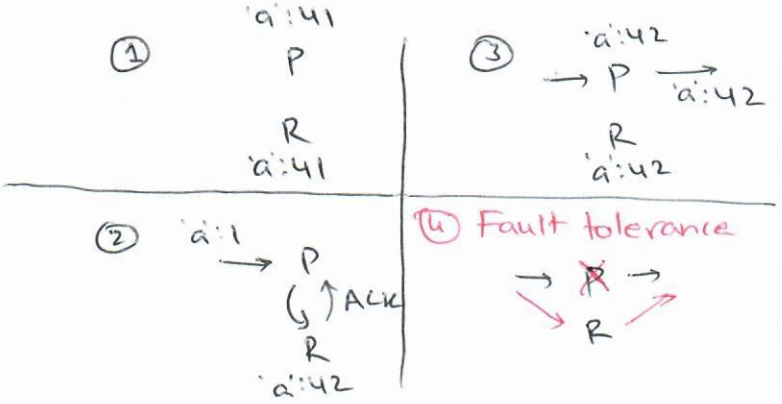
✗ Writing to durable storage is slow

Ideas? Run every stateful task as "replicated state machine" ③



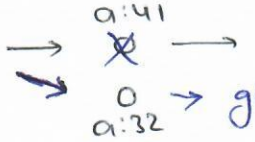
Primary and replica have uncorrelated failure (diff. racks/DC)

General idea of replicated state machine ③

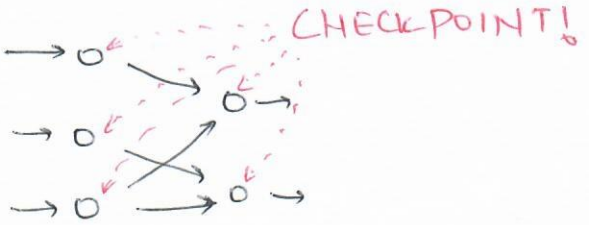


Problems ③

- Require 2x resources. Costly
- No straggler mitigation
- Runs as slow as slowest replica
- Replica not allowed to fall behind



Synchronous Global Checkpoints ③



- Stop consuming NEW messages
- Finish processing current message
- Save all channels
- Save all state

UNPAUSE

Fault Tolerance ③

- ALL workers/channels revert to last checkpoint (rebuild state)
- Source (tweets) cursor is also reverted

→ Problems ③

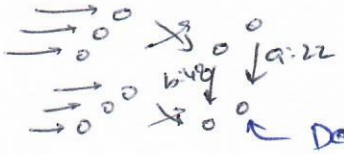
- Checkpoints stop the world
- Expensive fault tolerance

} Freshness :(

- Straggler mitigation
  - During checkpoint: Replace
  - Otherwise: Just like fault tolerance
- Cost is OK!

Spark streaming ③

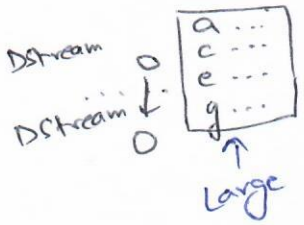
- ✓ Break streaming into mini-batches
- ✓ Pull state out



- ✓ Fault tolerance, straggler mitigation

Problems ③

* Not truly real-time.
  - Adds 0.5-2s delay in discretizing
* Large states?
  - Millions of words



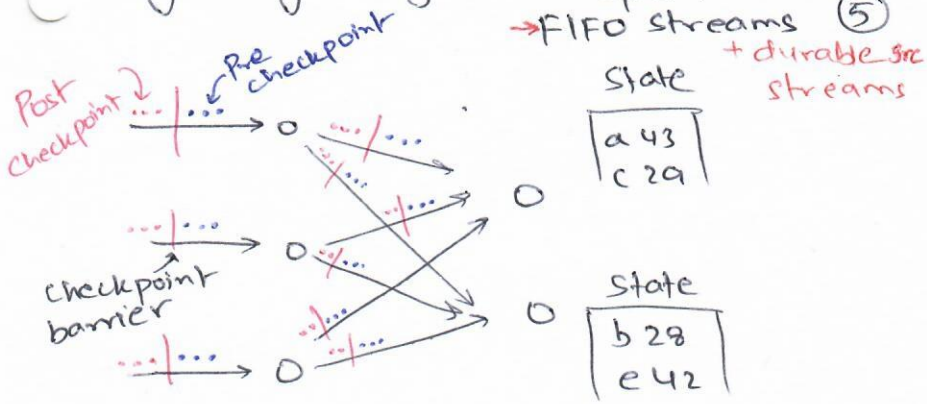
How to do FT? ①

Global sync. checkpoints
- Slow. Stops the world. Latency↑

Replicated state machines
- Cost ↑

Light weight async checkpoints (5)

→ FIFO streams + durable src streams



Chandy-Lamport Algo

Steps after recieving a barrier (3)

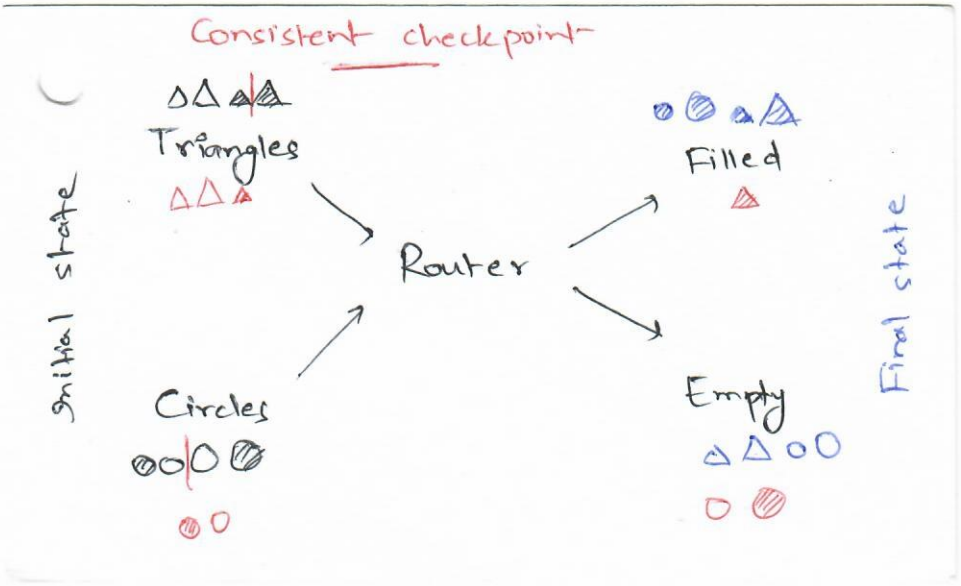
1. Wait for barrier on all input stream
2. Checkpoint local state
3. Put barrier on all output streams
4. Continue processing inputs

Proof of correctness (3)

- Each worker checkpoints when they've processed

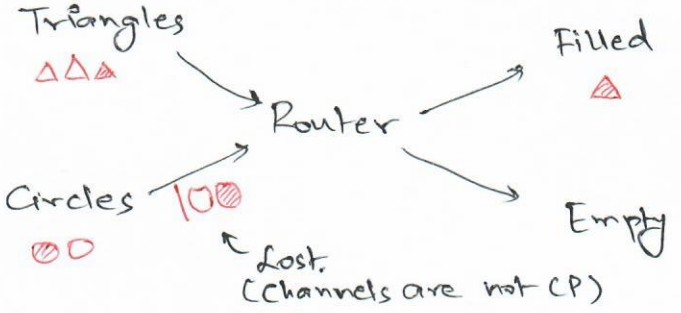
ALL pre-checkpoint messages

and NO post-checkpoint messages



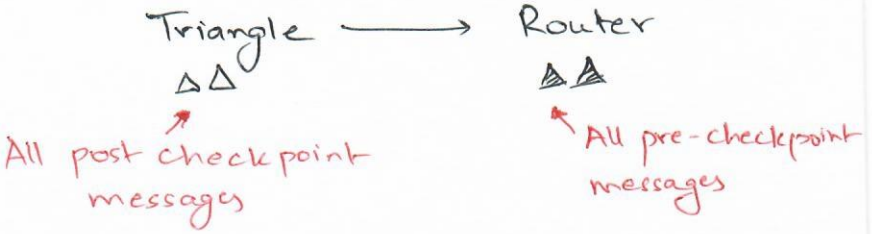
Questions.

* What if Router doesn't WAIT for ALL BARRIERS? Inconsistent checkpoints



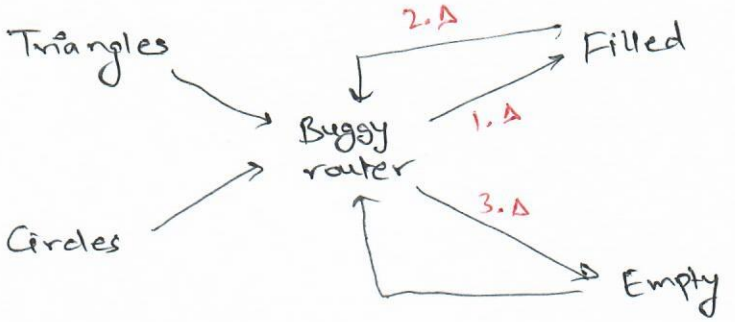
Do we need to checkpoint channels?

No. (for Acyclic graphs)



Source streams need to be rewinded

What about cycles?



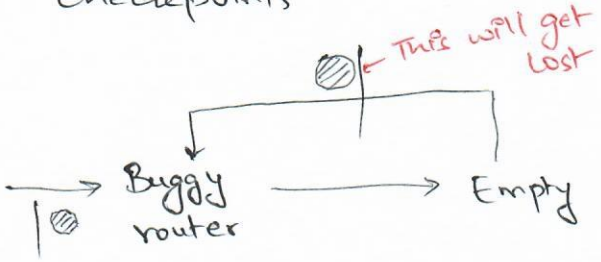
When should router checkpoint?

Can't wait for Barrier ON ALL input streams

Blocks forever due to back edge

sol. Block only on forward edges

This can produce inconsistent checkepoints



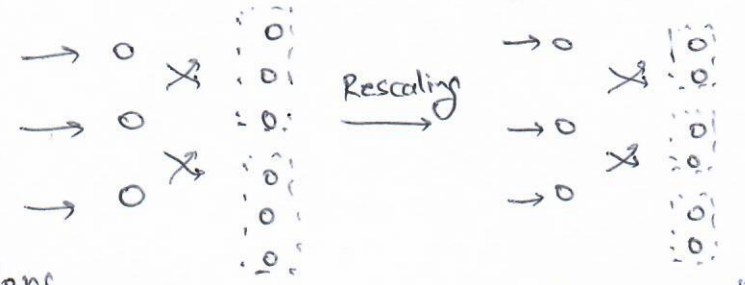
Sol: Store back edge channel messages

Further optimizations (3)

- Incremental checkpoints
- Fork checkpointing process, resume

Flink + RocksDB

Load Imbalance (3)



Options
- VM migration
- Pick up state from last checkpoint

"key groups" in Flink