Imperial College London

LSDS
Large-Scale Data & Systems Group

Virtualization cont.

Jana Giceva
jgiceva@doc.ic.ac.uk

Department of Computing
Imperial College London
http://lsds.doc.ic.ac.uk

Fall 2018

Virtualizing resources

Need to virtualize: (1) the CPU, (2) memory, (3) storage and (4) network I/O.

Tuesday's focus was on CPU virtualization!

Today we focus on memory, storage and network I/O.

Virtualizing Memory in VMs

Recall basic address translation

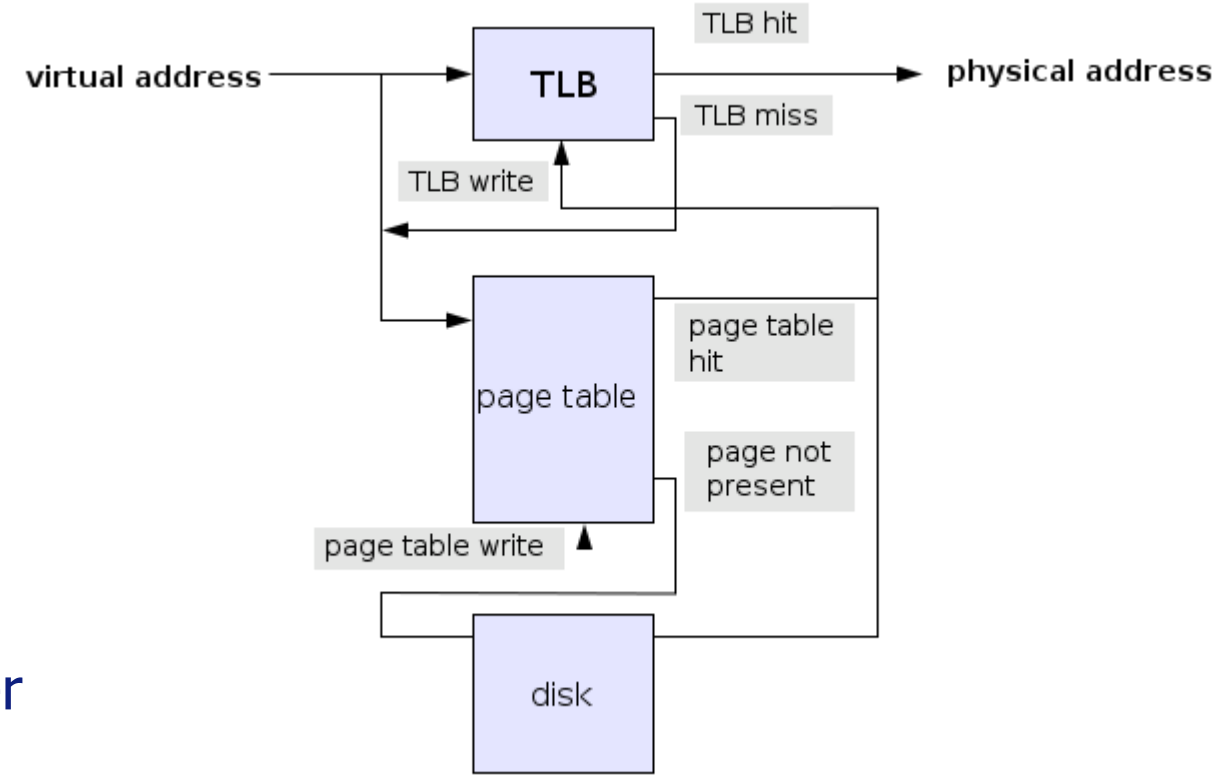
Physical address to virtual address translation

Translation stored in page tables

Translation caches in TLB caches

- TLBs filled by a hardware page table walker

Root of page table stored in `cr3` register



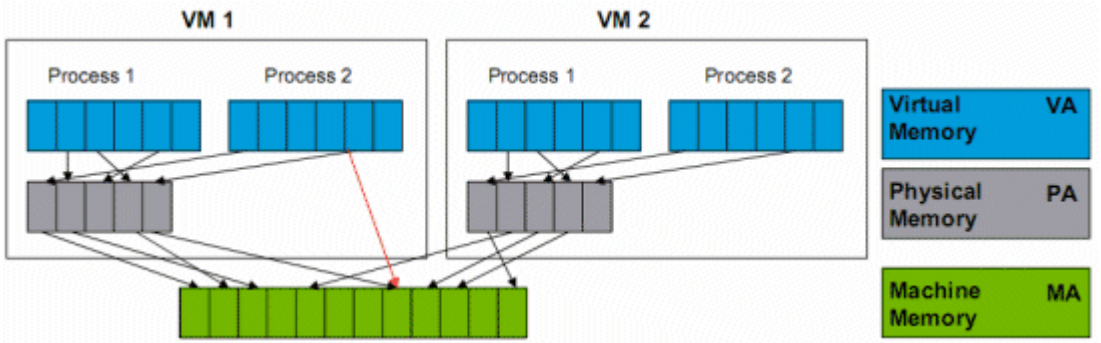
TLBs flushed based on `invlpg` instruction and context switch

Requirements for virtualizing memory in VMs

We need to do two translations:

1. Guest Virtual Address → Guest Physical Address

2. Guest Physical Address → Host Physical Address



We can store only one translation in the TLB

- Guest Virtual Address -> Host Physical Address

We store the translations (since only a few will fit in the TLB) in a "shadow" page table

- Index is the same as on the virtual machine: guest virtual address
- Output: host physical address

Shadow Page Tables – page fault flow

1. Accessing Guest Virtual Address causes a page fault
   - User → Guest OS
2. Walk page table in software to identify Guest Physical Address
3. If required, allocate "guest physical page" for faulting address
   - Return back to the user. Guest OS → User
   - When user tries to access address, it will fault again this time to the hypervisor
4. Translate Guest Physical Address to Host Physical Address
   - Do this for each guest physical address involved in page table
5. Allocate Page for Host Physical Address if required
6. Update shadow page table

} Hidden page faults → main overhead for memory virtualization

7. Install entry in TLB
   - Hypervisor → User

Shadow Page Tables – Overhead

Every time a guest tries to update its own page table, we must trap into the hypervisor

- The shadow page table must be updated
- Accomplished by marking guest page table as read-only, a write will generate a fault
- When is the page table modified?
  - To add page table entries (e.g., upon page fault)
  - To remove page table entries (e.g., upon unmap())
  - To change protection bits for a page (e.g., make mmap page shared, etc.)

Page table access/dirty bits increase the overhead

- Setting these bits in guest page table requires trapping to hypervisor

Every process in the guest OS has its own address table

- Every process also needs a shadow page table!
- Doubles the memory requirements for page tables ☹

Pros: no HW support
Cons: complex and can be slow due to overheads

Hardware Support for Memory Virtualization

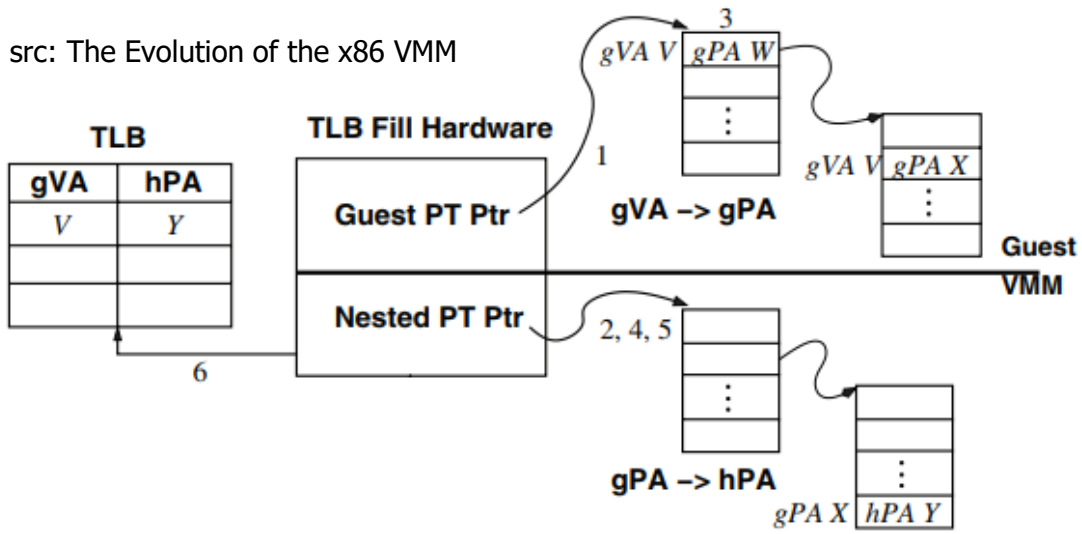
Intel:

- Extended Page Tables (EPT)

AMD:

- Nested Page Tables (NPT)

src: The Evolution of the x86 VMM



Intel's EPT does not support dirty bits.

TLB has address translation for different VMs, tagged with different VPIDs (ASIDs)

- No need to flush TLB when you switch VMs.

No need for hypervisor involvement

HW will walk guest page table, then host page table, and install entry in the TLB

Hypervisor involved only on page faults (why?)

Shadow pages vs. EPT

Both techniques maintain extra page tables

Key difference with Shadow Page Tables:

- Hypervisor maintains guest physical address to host physical address
- Shadow page tables have guest virtual to host physical translation
- CR3 register will point to guest physical address that is the root of guest page table

When using Shadow Page Tables, the hypervisor needs to be involved in:

- Modifications to page tables
- Page faults
- Context switches
- Invalidating page Table Entries

When using EPT:

- All the overhead is gone
- But page table walks are still more expensive – as we need to walk two page tables instead of one

Overall, EPT is much faster than Shadow Page Tables

Para-virtualization

Xen statically divides up memory between the virtual machines
Page tables are registered with the MMU

Design decisions:

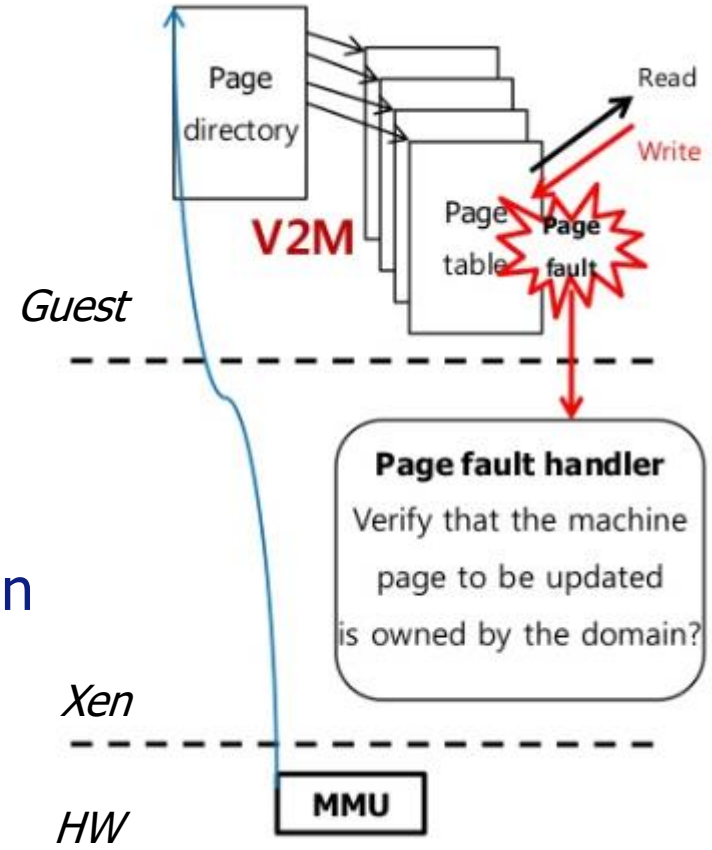
- Guest OSes allocate and manage the hardware page tables, with minimal involvement from Xen
- Xen exists in the 64MB section at the top of every AS, to avoid a TLB flush when entering and existing the hypervisor

VMs have read-only access, each PT update is verified by the Xen

- To ensure that pages in the page tables are not written by any application

Guests can batch updates to amortize the cost of going to Xen

One level translation speeds up workloads significantly → big advantage of para-virtualization



src: Memory Virtualization and Management – Hwanju Kim (KAIST)

Virtualizing network IO in VMs

Recall – how a NIC driver works

Transmit path:

- OS prepares a packet to transmit in a buffer in memory
- Driver writes start address of buffer to register X of the NIC
- Driver writes the length of the buffer to register Y
- Driver writes "1" (GO!) into register T
- NIC reads packet from memory addresses [X,X+Y) and sends it over the wire
- NIC sends interrupt to host (TX complete, next packet please)

Receive path:

- Driver prepares buffer to receive packet into
- Driver writes start address of buffer to register X
- Driver writes length of buffer to register Y
- Driver writes "1" (READY-TO-RECEIVE) into register R
- When packet arrives, the NIC copies it into memory at [X, X+Y)
- NIC interrupts host (RX)
- OS processes packet (e.g., wake up the waiting process)

Network I/O virtualization – emulation

Emulation

- Implement at virtual NIC (vNIC)
- vNIC's registers are variables in Hypervisor's memory
- Memory is write-protected (Hypervisor reacts based on the values being written)
- Interrupts are injected by hypervisor to guest

Pros

- Unmodified guest (it already has a driver)
- Can only use one device → robust

Cons

- Very slow (trap on every register access and there are many)
- The Hypervisor needs to emulate complex hardware

Example hypervisors: QEMU, KVM, VMWare

I/O virtualization – paravirtualization I

Paravirtualization

- Add virtual NIC driver into guest (front-end)
- Implement the vNIC in hypervisor (back-end)
- Everything works just like in the emulation case …
- … except: protocol between front-end and back-end, which is made efficient.

Protocol in emulation case:

- Guest writes registers X, Y, waits some ns and writes to register T
- → Hypervisor **infers** guests wants to transmit packet

Protocol in paravirtualization case:

- Guest does a hypercall, passes it start address and length as arguments;
- → Hypervisor **knows** what it should do

I/O virtualization – paravirtualization II

Paravirtual protocol can be high-level:

- E.g., ring of buffers to transmit (so NIC does not stay idle after each transmission) and independent of any particular NIC registers

Pros

- Fast
- Optimized for virtualization: minimize the number of VM←→ Hypervisor switches
- No need to emulate any physical device

Cons

- Requires guest to have a driver (i.e., the guest must be updated/modified)

I/O virtualization – direct assignment

Direct device assignment

- "Pull" NIC out of the host, and "plug" it into the guest
- Guest is allowed to access NIC registers directly, no hypervisor intervention
- Host can't access the NIC anymore

Pros:

- As fast as possible!

Cons:

- Need NIC per guest
- Plus one for the host

Example hypervisors: KVM, Xen, VMware

I/O virtualization – HW support (IOMMU)

IOMMU (I/O memory management unit)

- I/O devices (like our NIC) perform DMA operations, which can access memory on their own

Traditionally, devices used physical addresses to do so

This is a serious problem in a set-up where multiple untrusted guests are running simultaneously, sharing the same machine

- What if a guest is malicious?
- What if the device driver is buggy?
- → kills direct driver assignment
- Also what if device is legacy and can use only 32bit addresses, yet the physical memory is bigger

IOMMU solves this problem

- It allows the hypervisor to arrange things such that device use IOVAs (I/O virtual addresses) instead of Pas for their DMA operations
- Like the MMU, the IOMME knows how to walk the table and has a IOTLB
- Unlike the MMU, an I/O page fault is not tolerated → DMA-related memory must be pinned

I/O virtualization – HW support (SR-IOV)

SR-IOV

- The ability of a device to appear to software as multiple devices
- Single Root I/O virtualization (SR-IOV)
- Contains a physical function controlled by the host, used to create virtual functions
- Each virtual function is assigned to a guest (like in direct assignment)
- When a packet arrives for a specific VM, the network function of that VM is responsible for processing the packet, without interference from other VMs
- Each guest thinks it has full control of the NIC, accesses registers directly
- The NIC itself does multiplexing/demultiplexing of the traffic
- DMA the packet directly to the memory of the Guest OS (using VTD technology)

Pros

- As fast as possible
- Need only one NIC (as opposed to direct assignment)

Cons

- Requires new hardware

Intel I/O Acceleration Technology (IOAT)

Suite of hardware features to increase performance (throughput, scalability, etc).

1. QuickData Technology
   - Enables data copy by the chipset instead of the CPU to move data more efficiently through the server
2. Direct Cache Access (DCA) – see DDIO on next slide for more details
   - Allows I/O device to place data into CPU caches directly, reducing cache misses and improving latency
3. Extended Message Signaled Interrupts (MSI-X)
   - Distribute I/O interrupts to multiple CPUs and cores, improving CPU utilization
4. Receive Side Coalescing (RSC)
   - Aggregates packets from the same TCP/IP flow into one larger packet, reducing per-packet processing times
5. Low latency interrupts
   - Tune interrupt interval times depending on the latency sensitivity of the data, to improve efficiency

Intel Data Direct I/O (DDIO)

Enables I/O directly in and out of the processor's last level cache (LLC)

Neither the driver nor the NIC hardware need to do anything – the support is integrated in the platform, invisible to software

Motivated by the introduction of 10Gb Ethernet, IB, etc.

Very exciting for performance

- Increased bandwidth, reduced power consumption, and reduced latency
- I/O bound workloads can see dramatic performance benefits and reduced power consumption
- Latency-sensitive workloads will see latency benefits and much higher transaction rates
- Non I/O bound workloads in a DC will primarily benefit from reduced power usage

Packet processing game

10 Gbps delivers 14.8M packets per second (assuming 64 byte packets and 20 byte pre-amble)

That gives us 67 ns to process a single packet

67ns is about 200 cycles on a 3GHz processors → not a lot

For comparison:

- A cache miss is 32 ns
- L2 cache access is 4.3 ns
- L3 cache access is 7.9 ns
- Atomic lock operation is 8.25 ns (16.5 for unlock too)
- System call is 41.85 ns
- TLB miss → several cache misses

Need to batch and process packets

What to avoid: (1) a context switch (>1000 ns), and (2) page fault (>1000ns)

Data Plane Development Kit (DPDK)

Open-source project with contributions from Intel and others

- C code (CGG 4.5.x or later)
- Linux kernel 2.6.34 or later
- With kernel boot option isolcpus

Set user-space drivers and libraries for fast packet processing

- 10 or 40 Gb NICs
- Can handle 11x the traffic of the Linux kernel

Lightweight, low level, performance driven framework

All traffic bypasses the kernel (avoids interrupts)

Started with Intel x86, now supports other architectures like IBM Power 8

DPDK – What do you get?

UIO drivers

PMD per hardware NIC:

- PMD (Poll Mode Driver) support for RX/TX (Receive and Transmit)
- Mapping PCI memory and registers
- Mapping user memory (e.g., packet memory buffers) into the NIC
- Configuration of specific HW accelerations in the NIC
- Non-preemptable

User space libraries:

- Initialize the PMDs
- Threading (builds on pthread) – but not thread safe
- CPU management
- Memory management (physical memory allocation (mmap) of huge pages only!)
- Hashing, scheduler, pipelining (packet steering), etc.
- High performance support libraries for the application

DPDK – architecture overview

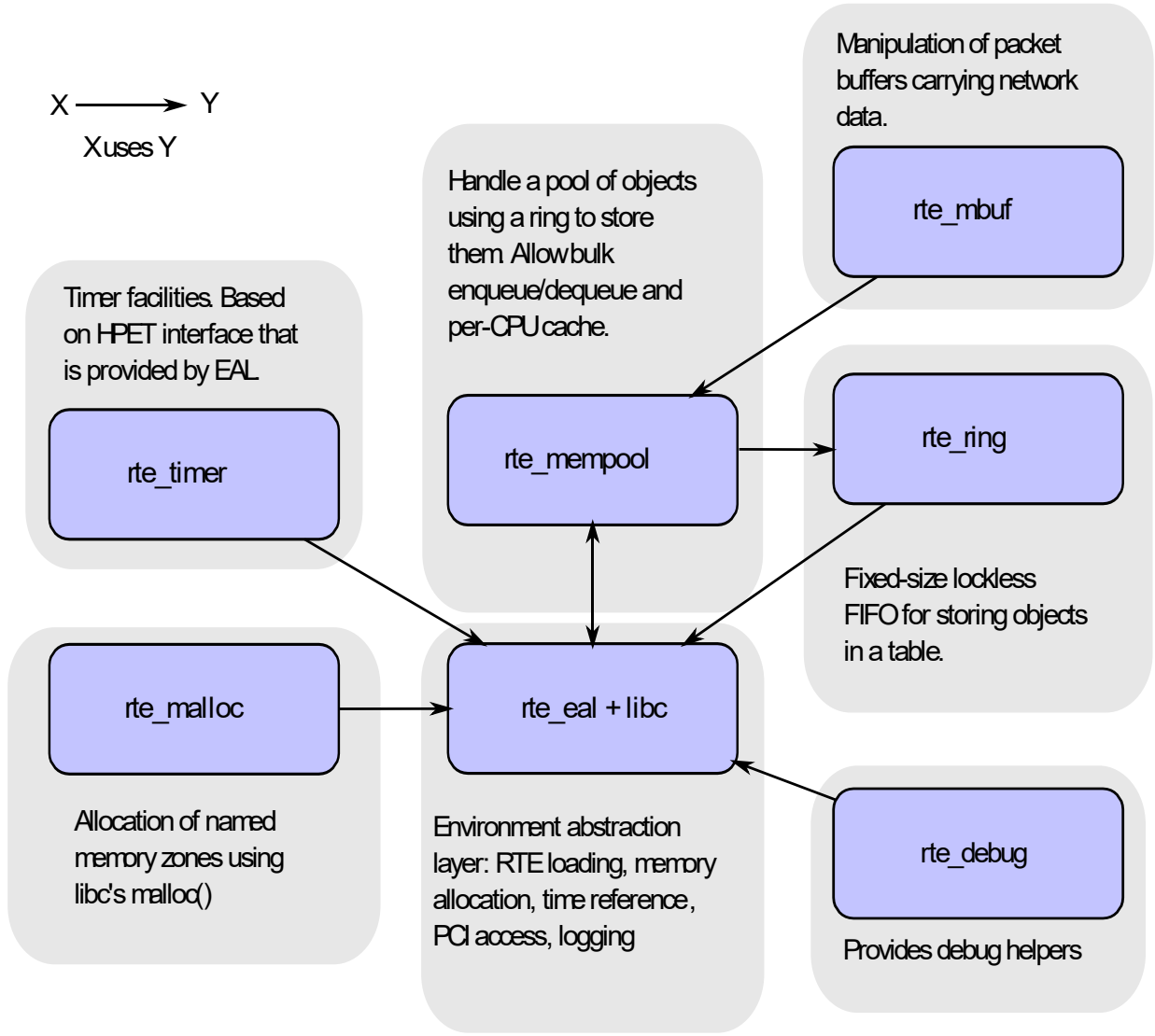
Core components – set of libraries that provide all that is needed for high performance packet processing.

1. Ring Manager (librte-ring)

- Lockless MPMC FIFO API in a finite size table
- Used by the Memory Pool Library
- As general communication mechanism between cores and/or execution blocks.

2. Memory Pool Manager

- Allocating pools of objects in memory
- Each pool uses a ring to store free objects
- NUMA and cache optimizations



DPDK – architecture overview

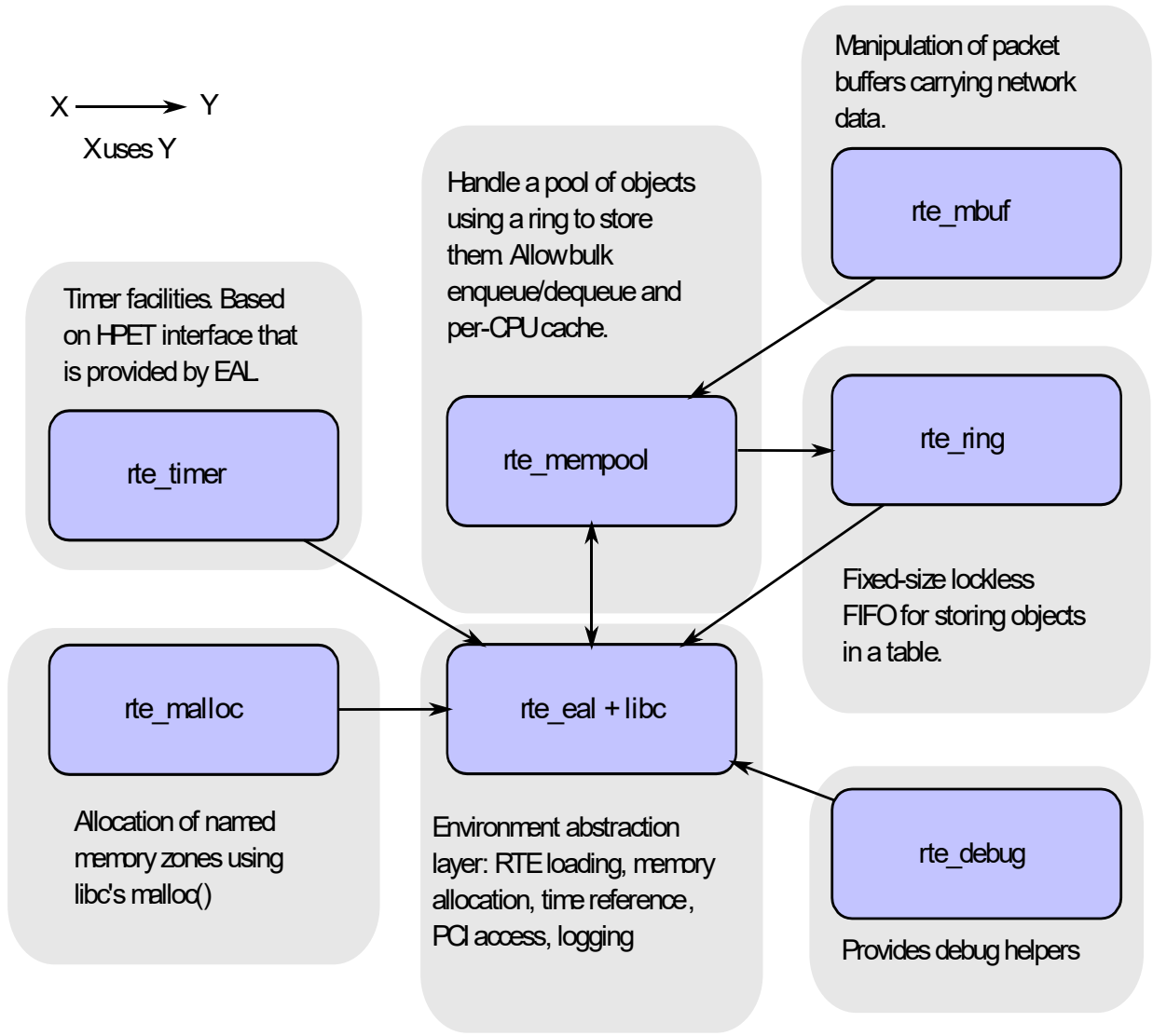
Core components – set of libraries that provide all that is needed for high performance packet processing.

3. Network Packet Buffer management (librte_mbuf)

- Create and destroy buffers for storing messages to be used by the DPDK application
- Created at start-up and stored in mempool

4. Timer management (librte-timer)

- Support to execute a function asynchronously



What does basic forwarding do?

1. DPDK init – initialize the environment abstraction layer (EAL)
2. Get all available NICs
3. Initialize packet buffers (create memory pool)
4. Initialize the NICs (4.1 their ports, 4.2 queues (Rx queue, Tx queue)
5. Start getting packets

PMD loop for receiving and forwarding packets

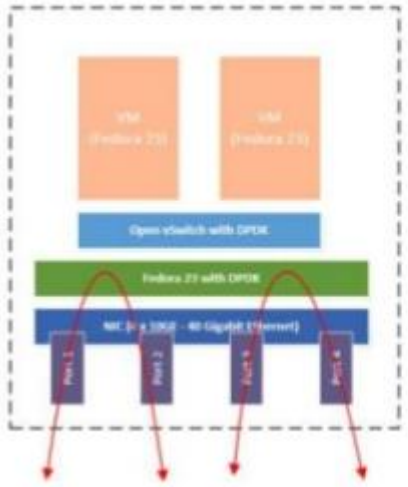
- Get bursts of Rx packets from first port in port-pair
- Send packets of Tx to second port in port-pair
- Free any unsent packets
- Do this in a loop

Performance Evaluation (reported 2016)

Set-up:

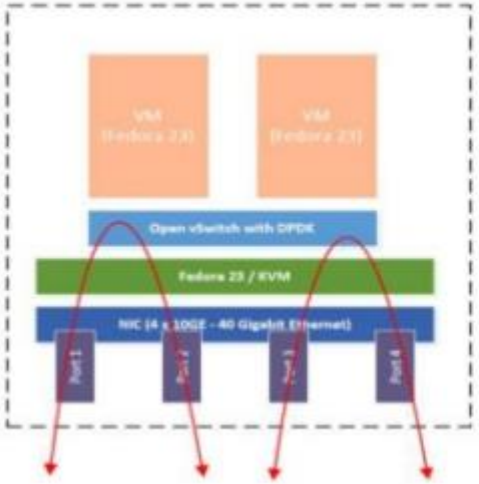
- 4X40Gb ports
- E5-2695 V4 2.1Ghz Processor
- 16X1GB Huge Pages, 2048X2MB Huge Pages

PHY-PHY



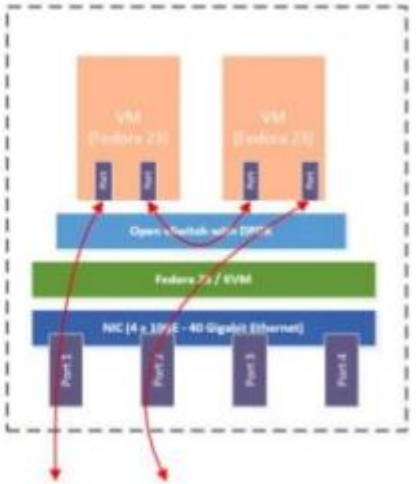
22 Mpps per core

PHY-OVS-PHY



11 Mpps per core

VM-VM



2 Mpps per core

Intel Open Network Platform Release 2.1, Performance Test Report (March 2016)

DPDK Pros and Cons

Advantages:

- Excellent performance (both for single and multi-cores)
- Great virtualization support
- Active community (from 2012)
- Active popular products
  - OVS-DPDK is the main solution for high speed networking in openstack
  - 6WIND
  - Trex

Disadvatages:

- Security
- Isolated ecosystem – hard to use Linux kernel infrastructure
- Requires modified code in applications (specific API, can't interact transparently with Linux processes)
- Requires Huge Pages

Virtualizing Storage

Virtualizing Storage

Simplest approach:

- Let the VM take a single device or a partition (raw device mapping RDM)
- But, this wastes resources if the VMs do not fully utilize their partitions

Thus, storage is virtualized by emulating multiple logical devices from a single physical device

For example, a VMDK (Virtual Machine Disk) file represents a virtual disk as seen by the virtual machine

Operations on the VMDK are translated into ops on the underlying storage device.

VMDK benefits

You can overprovision your virtual disks.

- For example, you can provide 5x 1TB virtual disks on top of a single 1 TB storage device.
- This works as long as the VMs do not fully utilize their space.

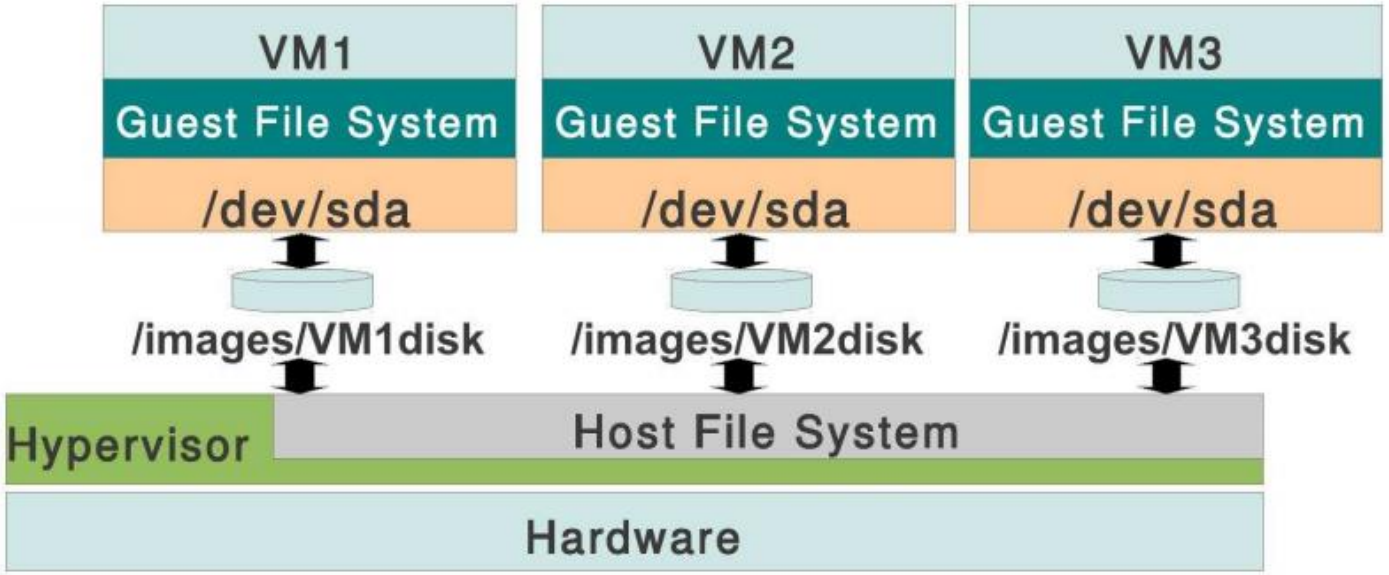
The VMDK starts out completely unallocated.

As the VM writes to a disk block, it populates the VMDK file.

You can do deduplication among multiple VMDKs since they are just files on the host.

Nested File System in a Virtualized Environment

1. File system inside VM
   - e.g., ext4 on virtual disk
2. Virtual Disk inside the VM
3. File system on host
   - VMDK file sits on this file system
4. Physical storage device on the host



src: Understanding Performance Implications of Nested File System in a Virtualized Environment – FAST'12

Nested File System – Problems

All these layers introduce a number of performance problems:

E.g., double journaling:

- 1. Journaling inside the VM, and
- 2. on the VMDK file on the host.

If you are updating a inode in the guest file system, it is journaled (2x IO)

If the host file system also uses journaling, the metadata of the VMDK is also journaled (3x IO)

File systems make assumptions about the storage device →

if not true, then optimizations actually reduce the performance.

Combination of gFS and hFS can hurt performance

The combination of the file system on the guest and the file system on the host is very important:

- The wrong combination can reduce throughput by 67% of max (reiserfs on ext2)
- When ext2 runs on top of ext3, throughput reduced by 10%
- When ext3 runs on top of ext3, throughput reduced by 40%

For read-only workloads, stacking the file system actually helps

- Why?
- Read-ahead issued by the host file system

Ideally, host should not have smarts: use act as a simple on-demand allocator for guest file system.

Nested Virtualization

Nested Virtualization

Goal: run unmodified hypervisors on top of a hypervisor, achieve close to single-virtualization level performance

Project: Turtles (OSDI'10) from IBM Research – Haifa

The Turtles Project: Design and Implementation of Nested Virtualization

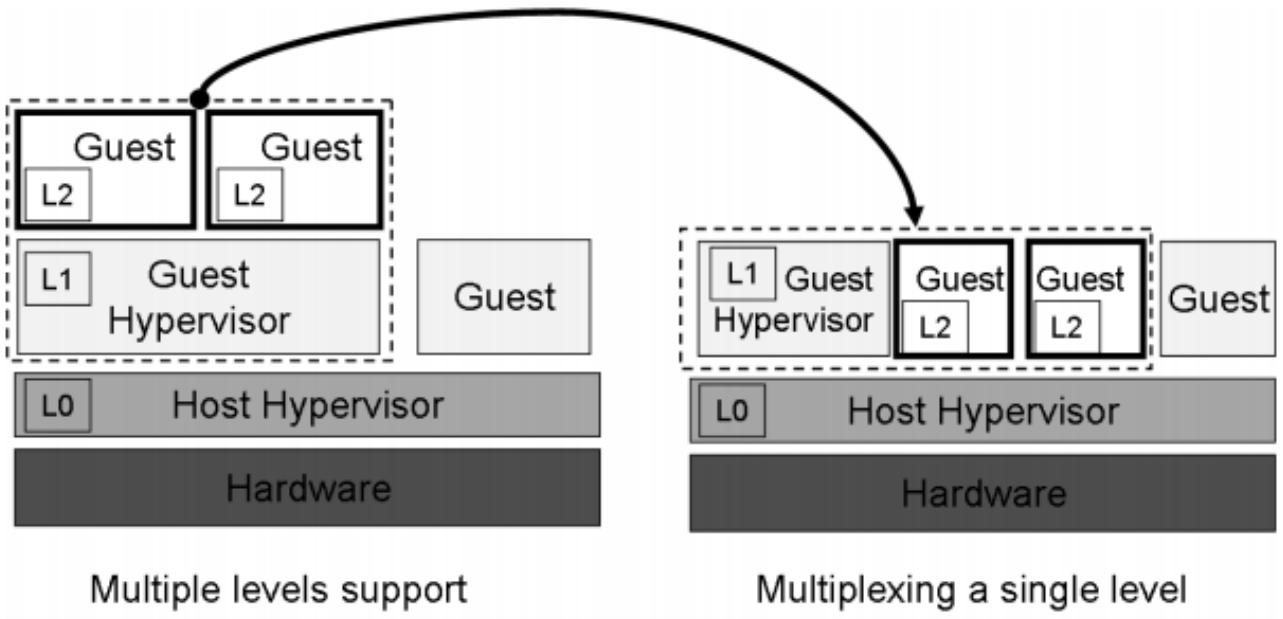
Muli Ben-Yehuda[†] muli@il.ibm.com
Michael D. Day[‡] mdday@us.ibm.com
Zvi Dubitzky[†] dubi@il.ibm.com
Michael Factor[†] factor@il.ibm.com
Nadav Har'El[†] nyh@il.ibm.com
Abel Gordon[†] abelg@il.ibm.com
Anthony Liguori[‡] aliguori@us.ibm.com
Orit Wasserman[†] oritw@il.ibm.com
Ben-Ami Yassour[†] benami@il.ibm.com
[†]*IBM Research – Haifa*
[‡]*IBM Linux Technology Center*

Interesting fact: this is achieved without hardware support

Based on the KVM hypervisor

Nested Virtualization: key technique

Multiplexing multiple levels of virtualization onto a single level of hardware support



1. The host hypervisor emulates the hardware support (VMX) to higher level hypervisors

2. Each hypervisor emulates VMX to the hypervisors above it

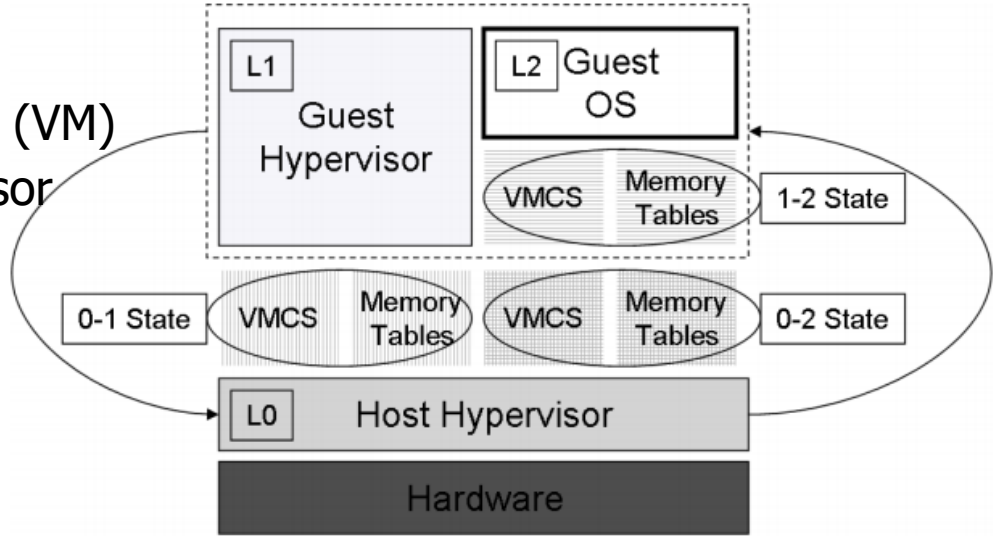
3. To L0 hypervisor, both L1 hypervisor and L2 guest are similar: both are seen as virtual machines.

4. A trap in L1 or L2 goes to L0 hypervisor. The L0 hypervisor then forwards the trap to the correct hypervisor.

Nested Virtualization: key technique cont.

Review of hardware support

- Separate registers for root mode (hypervisor) and guest mode (VM)
- Control data for specifying what events should trap to hypervisor and for guest to indicate to hypervisor why it trapped
- The VMCS structure is maintained by hardware



When L1 wants to run a VM

- It runs the vmlaunch command
- This traps to L0
- L0 then runs the L2 VM specified in the L1 command

L0→L1 VMCS is maintained by hardware

L1→L2 VMCS is maintained by L1; this is trapped, and L0→L2 VMCS is calculated and used

Virtualizing the MMU: Multi-dimensional paging

L0 and L1 hypervisors might each decide to use shadow page tables for memory virtualization.

Shadow paging is too slow for use in nested virtualization.

L0 exposes the EPT capability to L1 (i.e., allows L1 to construct an EPT page table)

- Then compresses the EPT page tables from L2 to L1 and L1 to L0 into one

Page translation in L2:

- L2 virtual to L2 physical (its own page table)
- Where is L2 physical? Faults in L0
- L0 forwards this to L1
- L1 constructs EPT table for mapping L2 to L1
- When L1 is done, it does vmreturn to L2 (this traps to L0)
- L0 makes page table memory of L1 page table as read only
  - L0 will know if L1 tries to change the page table
- L0 constructs compressed EPT table L2 to L0
- L2 returns into L0

Virtualizing Devices

Allow L2 to directly access device, bypassing L1 and L0

DMA is done by IOMMU which translate between guest physical and host physical

- Problem: only one level of translation is supported

L1 emulates IOMMU for L2

More details in the Turtles paper.

Security and VMs

Security when working with VMs

Security depends upon a number of manual actions: e.g., patching the machine

It's hard to do in VMs because of how many VMs there might be in the organization and how easy it is to spin up new VMs

Hard to understand the state of the network

- VMs appearing and disappearing all the time
- A vulnerable VM may be suspended currently and miss the security patch
- If you checkpoint a VM and roll back to its earlier state – may lose the security patch update
- A number of VMs may have OSes at different update points → diversity in the network
- Typically a real machine is identified by its MAC address – what to do for VMs?

Problems introduced by Virtualization

In a non-virtualized setting, the OS trusts the hardware

In virtualized settings, the VMs trust the VMM. However, is the VMM as trust-worthy as the hardware?

Solution: Trusted Platform Module (TPM)

- Can attest to integrity of software components
- Outside the CPU (or on CPU's Trusted Execution Environment TEE – SW, so can have bugs)

Virtual Machine Introspection (VMI)

- For monitoring the runtime state of a system-level VM. Helpful for debugging and forensic analysis.

Encryption keys in software can be read

Problems introduced by Virtualization (cont.)

Attack: VMM Rootkits

- Transparently inserting a VMM under an OS

Example from a survey paper:

"Company A has a virtual server in an outsourced datacenter that undertakes financial transactions. Depending upon the contract with the data-center, it is likely that the datacenter does not have permission to view or alter any transactions undertaken (based on least need-to-know principles). However, because Company A does not control the underlying VMM, it has no way to ensure that the VMM has not altered transaction details or recorded credentials, a potential problem in many ways, as any local audit trails can be similarly compromised."

Important to know: VMs do not fully isolate!

"Hey, You, Get Off my Cloud" – Ristenpart et al. (CCS'09)

"Using the Amazon EC2 service as a case study, we show that it is possible to map the internal cloud infrastructure, identify where a particular target VM is likely to reside, and then instantiate new VMs until one is placed co-resident with the target.
We explore how such placement can the be used to mount cross-VM side-channel attacks to extract information from a target VM on the same machine"

Attacker can:

- Determine where the cloud infrastructure instance is located
- Can easily check if two instances are co-resident on the same machine
- Launch instances that will be co-resident with other user's instances
- Exploit cross-VM leakage information once co-resident

References

1. The Evolution of an x86 Virtual Machine Monitor
2. Xen and the Art of Virtualization (SOSP'03)
3. Understanding Performance Implications of Nested File Systems in a Virtualized Environment (FAST'12)
4. The Turtles Project: Design and Implementation of Nested Virtualization (OSDI'10)
5. When Virtual is Harder than Real: Security Challenges in VM based Computing Environments (HotOS'05)
6. Hey, You, Get Off My Cloud: Exploring Information leakage in Third-Party Compute Clouds (CCS'09)

Slides also influenced by material from Ariel Waizel, Dan Tsafrir, Muli Ben-Yehuda, Patrick Kutch, Vijay Chidambaram, DPDK.org, Intel, Microsoft, Oracle