Advanced Computer Networks
263-3501-00

Network I/O Virtualization

Spring Semester 2018
Today

- TCP in the data center
- Remote Direct Access Memory (RDMA)
- High-performance flow control
- Virtual machine networking
  - Para-virtualization
  - SR-IOV
  - IOMMU
Processor Clock Frequency Scaling Has Ended

- Moore’s Law continues in transistor count
- Industry response: **Multi-core** (i.e. double the number of cores every 18 months instead of the clock frequency (and power!))
Virtualization and Hypervisors

- VM1
  - Guest Application
  - Guest Operation System
- VM2
  - Guest Application
  - Guest Operation System
- VM3
  - Guest Application
  - Guest Operation System

Hypervisor

Hardware
Virtualization and Hypervisors

How does network access work?
1) **Application**: syscall, e.g., socket.write()
   - Trap into kernel

2) **OS driver**: issue PCI commands
   - Set up DMA operation

3) **NIC**:
   - transmit data
   - raise interrupt when done
Option 1: Full Device Emulation

- Guest app makes system call:
  - Trap into hypervisor
  - Hypervisor jumps back to guest OS

- Privileged instruction in guest OS
  - Trap into hypervisor
  - Hypervisor emulates instruction
  - After emulation, jump back to guest OS

- Advantage: no changes to the guest OS required

- Disadvantage:
  - Inefficient: Many privileged calls necessary per I/O operation
  - Complex

Diagram:
- Application (Ring 3)
- Guest OS (Ring 1)
- Traps
- Device Emulation
- Hypervisor
- Hardware
Historical X86 Virtualization Limitations

- Sensitive/privileged instructions when executed in ring 3 may have one of three outcomes:
  1) a fault occurs
  2) the process issues a trap indicating that it wants code in ring 0
  3) Nothing

- Silently failing sensitive instructions make it difficult to implement virtualization

- Solution:
  - Binary translation: translate problematic instructions in guest to instructions that actually trap
  - Hardware-assisted virtualization
Intel Virtualization Technology VT-x

- Introduces host and guest mode
  - Each with 4 privilege levels
- Protected instructions executed in guest mode, ring 0, generate faults that can be checked in host mode
Starting new guest = starting QEMU process

QEMU process interacts with KVM through ioctl on /dev/kvm to
  - Allocate memory for guest
  - Start guest in guest mode ring 0

I/O requests from guest OS trap into KVM (VM exit)

KVM on VM exits forwards requests to QEMU for emulation
  - Unless its a simple request then it forwards the request to the kernel
Option 2: Paravirtualization

- Guest OS aware that it is being virtualized
  - Runs special paravirtual device drivers
- Hypervisor cooperates with guest OS through paravirtual interfaces
- Advantage:
  - Better performance
  - Simple
- Disadvantage:
  - Requires changes to the guest OS
Paravirtualization with VirtIO

- VirtIO: I/O virtualization framework for Linux
  - Framework for developing paravirtual drivers
  - Split driver model: front-end and back-end driver
  - APIs for front-end and back-end to communicate
VirtIO and KVM

1) VirtIO-Net driver adds packet to shared VirtIO memory
2) VirtIO-Net driver causes trap into KVM
3) KVM schedules QEMU VirtIO Back-end
4) VirtIO back-end gets packet from shared VirtIO memory and emulates I/O (via system call)
5) KVM resumes guest
Vhost: Improved VirtIO Backend

- Vhost puts VirtIO emulation code into the kernel
  - Instead of performing system calls from userspace (QEMU)
Where are we?

- **Option 1: Full emulation**
  - No changes to guest required
  - Complex
  - Inefficient

- **Option 2: Paravirtualization**
  - Requires special guest drivers
  - Enhanced performance
Where are we?

- Option 1: Full emulation
  - No changes to guest required
  - Complex
  - Inefficient

- Option 2: Paravirtualization
  - Requires special guest drivers
  - Enhanced performance

Not good enough!
Still requires hypervisor involvement, e.g., interrupt relaying
Where are we?

- **Option 1: Full emulation**
  - No changes to guest required
  - Complex
  - Inefficient

- **Option 2: Paravirtualization**
  - Requires special guest drivers
  - Enhanced performance

- **Option 3: Passthrough**
  - Directly assign NIC to VM
  - No hypervisor involvement: best performance

Not good enough! Still requires hypervisor involvement, e.g., interrupt relaying
Passthrough / Direct Assignment

VM1
- Guest Application
- Guest Operation System
- Native Device Driver
- Safe HW IF
- Hypervisor
- Hardware
- Ethernet

VM2
- Guest Application
- Guest Operation System
- Native Device Driver
- Safe HW IF
- Hypervisor
- Hardware
- Ethernet

VM3
- Guest Application
- Guest Operation System
- Native Device Driver
- Safe HW IF
- Hypervisor
- Hardware
- Ethernet
Paravirtual vs Passthrough in KVM

**VM1**
- Application
- Guest OS
- VirtIO-Net Driver
- KVM module
- Hypervisor
- Real NIC
- tap
- vhost net
- tx

**VM2**
- Application
- Guest OS
- Physical Driver
- NIC exclusively assigned to VM2
- Real NIC

Hypervisor

Systems@ETH Zürich
Challenges with Passthrough / Direct Assignment

- VM tied to specific NIC hardware
  - Makes VM migration more difficult

- VM driver issues DMA requests using VM addresses
  - Incorrect: VM physical addresses are host virtual addresses (!)
  - Security concern: addresses may belong to other VM
  - Potential solution: let VM translate it's physical addresses to real DMA addresses
    - Still safety problem: exposes driver details to hypervisor, bugs in driver could result in incorrect translations

- Need a different NIC for each VM
Challenges with Passthrough / Direct Assignment

- VM tied to specific NIC hardware
  - Makes VM migration more difficult

- VM driver issues DMA requests using VM addresses
  - Incorrect: VM physical addresses are host virtual addresses (!)
  - Security concern: addresses may belong to other VM
  - Potential solution: let VM translate its physical addresses to real DMA addresses
    - Still safety problem: exposes driver details to hypervisor, bugs in driver could result in incorrect translations
  - Solution: Use an **IOMMU** to translate/validate DMA requests from the device

- Need a different NIC for each VM
  - Solution: **SR-IOV**, emulate multiple NICs at hardware level
Memory Address Terminology

- **Virtual Address**
  - Address in some virtual address space in a process running in the guest OS

- **Physical Address:**
  - Hardware address as seen by the guest OS, i.e., physical address in the virtual machine

- **Machine address:**
  - Real hardware address on the physical machine as seen by the Hypervisor
IOMMU

PCIe function (e.g. NIC)
IOMMU

VMM programs IOMMU with VM-physical to machine address translations

IOMMU

Memory controller

Main memory

PCIe function (e.g. NIC)
IOMMU

VMM programs IOMMU with VM-physical to machine address translations

IOMMU

Memory controller

Main memory

Guest OS programs NIC with VM-physical address of DMA

PCIe function (e.g. NIC)
IOMMU

VMM programs IOMMU with VM-physical to machine address translations

Guest OS programs NIC with VM-physical address of DMA

NIC issues a DMA request to VM physical memory

PCIe function (e.g. NIC)
IOMMU

IOMMU checks and translates to machine (real) address for transfer

VMM programs IOMMU with VM-physical to machine address translations

IOMMU

Memory controller

Main memory

Guest OS programs NIC with VM-physical address of DMA

NIC issues a DMA request to VM physical memory

PCIe function (e.g. NIC)
IOMMU

IOMMU checks and translates to machine (real) address for transfer

VMM programs IOMMU with VM-physical to machine address translations

Memory controller

Main memory

IOMMU

Guest OS programs NIC with VM-physical address of DMA

NIC issues a DMA request to VM physical memory

Memory controller accesses memory

PCIe function (e.g. NIC)

Memory controller accesses memory
SR-IOV

- Single-Root I/O Virtualization
- Key idea: dynamically create new “PCI devices”
  - Physical Function (PF): original device, full functionality
  - Virtual Function (VF): extra device, limited functionality
  - VFs created/destroyed via PF registers
- For Networking:
  - Partitions a network card's resources
  - Direct assignment of VF to VM to implement passthrough
SR-IOV in Action

- Guest Application
- Guest OS
- Physical Driver

- Guest Application
- Guest OS
- Physical Driver

- Guest Application
- Guest OS
- Physical Driver

- Hypervisor
- Physical Driver

- IOMMU

- PCI

- Virtual Function
- Virtual Function
- Virtual Function
- Physical Function

Virtual Ethernet Bridge/Switch

SR-IOV NIC
SolarFlare SFN6122F

127 VFs per port
Inter-VM communication

VM1
- Guest Application
- Guest Operation System

VM2
- Guest Application
- Guest Operation System

VM3
- Guest Application
- Guest Operation System

Hypervisor

NIC
Inter-VM communication

How does inter-VM communication work?
Switch in Hypervisor

Guest Application
Guest Operation System

Bridge/switch

NIC
Switched Vhost in KVM

- Advantage: low latency (1 extra software copy)
- Disadvantage: uses host CPU cycles
Switch Externally...

...either in

- **External switch:**
  - Simplifies configuration: all switching controlled/configured by the network
  - Latency = 2xDMA + 2hops
- **NIC**
  - Latency = 2xDMA
Controversial

- External switching in NIC or Switch
  - Extra latency
  - Reduces CPU requirements
  - Hardware vendors like it
  - Better TCAMs on the switch
  - Integration with network management policies

- Software switching in hypervisor
  - Lower latency
  - Higher CPU consumption.
  - CPU resources are generic and flexible
  - Easy to upgrade
  - Fully support OpenFlow
Container vs VMs

Containers are isolated but share OS and, where appropriate, bins/libraries.
Container Networking

Docker container are connected via a virtual docker Network (Eth bridge)
Moral

- Network interface cards traditionally are the “end point”
- Virtualization may add two more hops
  - Virtual switch in the NIC
  - Virtual switch in the hypervisor or docker namespace
- Inside of a physical machine increasingly resembles a network