Delivering low-latency communication in the Cloud

Anastassios Nanos, Nectarios Koziris

High-Performance Systems and Interconnects (HPSI),
Computing Systems Laboratory,
National Technical University of Athens
Github: `http://github.com/{HPSI,ananos}`
WWW: `http://cslab.ece.ntua.gr/~ananos`

OPERATIONAL PROGRAMME EDUCATION AND LIFELONG LEARNING
investing in knowledge society
MINISTRY OF EDUCATION, LIFELONG LEARNING AND RELIGIOUS AFFAIRS
MANAGING AUTHORITY
European Union European Social Fund
Co- financed by Greece and the European Union
NSRF 2007-2013

June 11th, 2013

Overview

1. I/O access in Virtualized Environments
   - Xen
   - KVM
2. Generic I/O operations
   - I/O options
   - Classes of I/O devices
3. High-performance Interconnects
   - Basic Concepts
   - Native networking frameworks
   - Networking frameworks in VM environments
4. Xen2MX
   - Xen2MX Architecture
   - Xen2MX evaluation
5. Conclusions
   - Summary

Introduction

Cloud computing paradigm

- less communication oriented,
- but still, distributed and decentralized

The Cloud

X-as-a-Service $\rightarrow$ HPC-as-a-Service

Introduction

Cloud computing paradigm

- less communication oriented,
- but still, distributed and decentralized

The Cloud

X-as-a-Service $\rightarrow$ HPC-as-a-Service

HPC in the Cloud

To efficiently execute applications, we need to optimize:

- CPU/Memory multiplexing
- I/O access

Introduction

Applications deployed in a native cluster

while !converge: compute $\rightarrow$ communicate

I/O: communication bottlenecks in application execution

- intermediate software layers:
  - copies, page table manipulation
  - interrupt/event handling

Introduction

Applications deployed in a native cluster

while !converge: compute → communicate

I/O: communication bottlenecks in application execution

- intermediate software layers:
  - copies, page table manipulation
  - interrupt/event handling

Interconnection frameworks

- MPI
- Infiniband, Myrinet etc. vs. Ethernet (Top500, gigE)
- TCP/IP Byte Transfer Layer (BTL)

Overview

1. I/O access in Virtualized Environments
   - Xen
   - KVM
2. Generic I/O operations
   - I/O options
   - Classes of I/O devices
3. High-performance Interconnects
   - Basic Concepts
   - Native networking frameworks
   - Networking frameworks in VM environments
4. Xen2MX
   - Xen2MX Architecture
   - Xen2MX evaluation
5. Conclusions
   - Summary

I/O Internals – Xen

Xen basics

- hypervisor – driver domain runs as a Linux guest
- split driver model (frontend/backend)

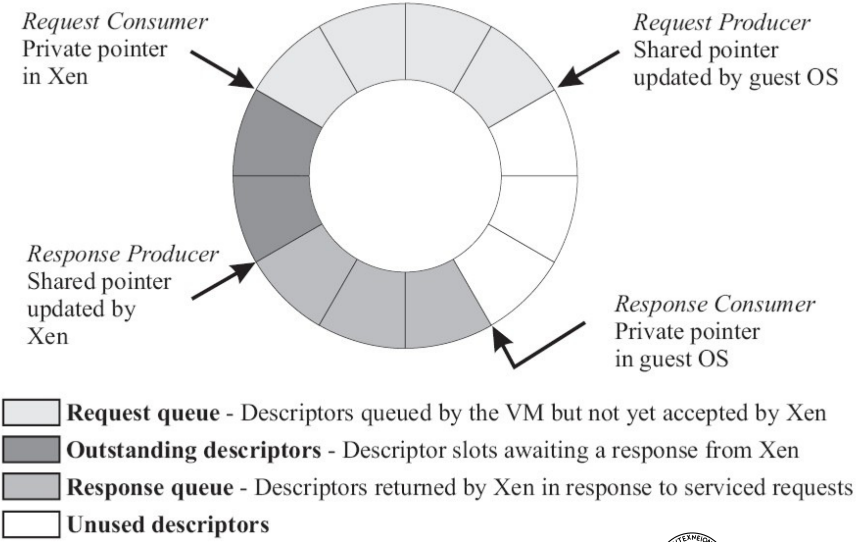
Xen – Event channels

- notify Guest/Host about a pending transaction
- easy to setup – bind to a specific "port"

Xen – Grant mechanism

- issue a page grant request
- the other end maps the grant (accept)
- this page is shared across the two domains

I/O Internals – Xen Ring buffers



I/O Internals – KVM

KVM basics

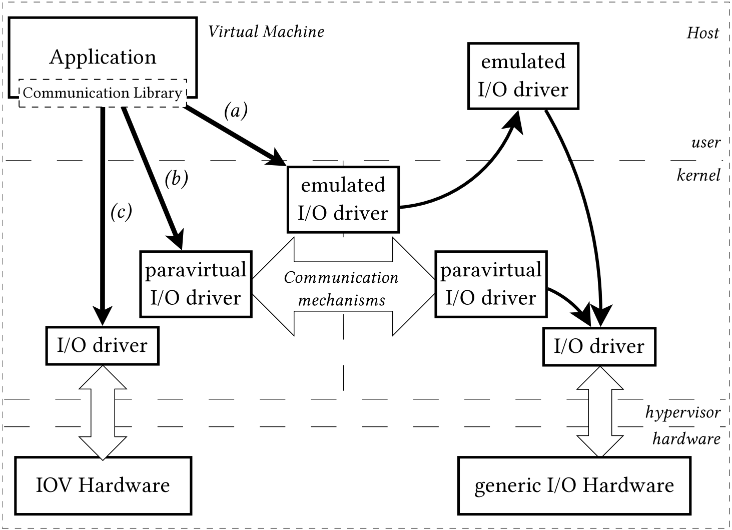
- Linux based (the hypervisor is the linux kernel)
- paravirtual setup: `virtio`

KVM – virtio ring

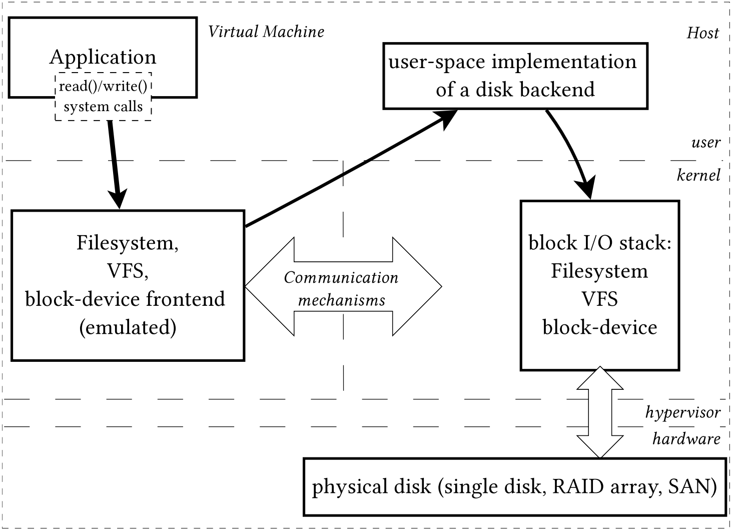
- virtqueues (simple queue to post buffers to be processed)
- virtio_ring:
  - descriptor array where the guest chains together length/address pairs
  - available ring where the guest indicates the ready-for-use chains
  - used ring where the host indicates the used chains
- "kicks" to notify Guest/Host (interrupts)

Overview

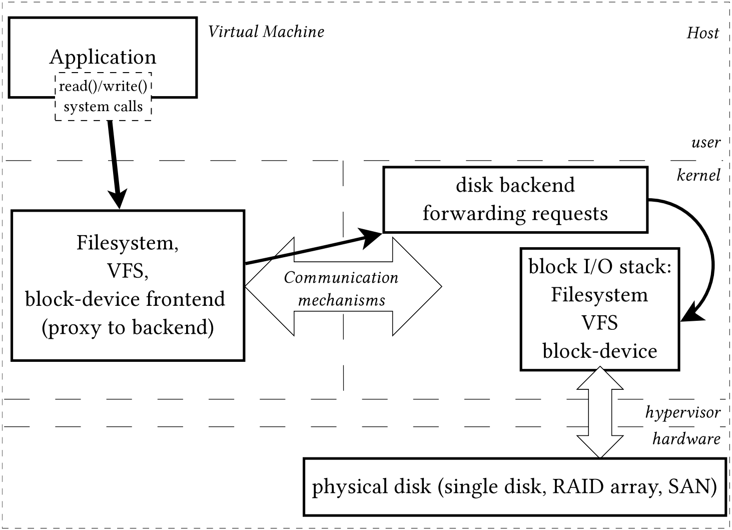
I/O options in a Virtualized Environment



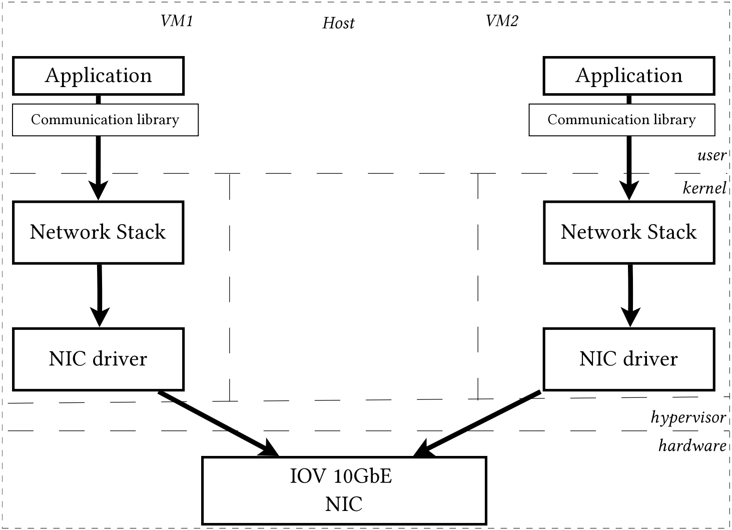
I/O options in a Virtualized Environment: emulation



I/O options in a Virtualized Environment: paravirtual



I/O options in a Virtualized Environment: IOV



I/O handling in popular open-source hypervisors

Parameters:

- device emulation:
  - totally unaware of the underlying platform – fully flexible! (migration, checkpointing, etc.) (+)
  - significant performance penatly (–)

I/O handling in popular open-source hypervisors

Parameters:

- device emulation:
  - totally unaware of the underlying platform – fully flexible! (migration, checkpointing, etc.) (+)
  - significant performance penatly (–)
- paravirtual drivers
  - needs extra effort to build the interface to the native driver (–)
  - scalable, as the interface to both VMs and the hardware is tailored to each driver class (+)

I/O handling in popular open-source hypervisors

Parameters:

- device emulation:
  - totally unaware of the underlying platform – fully flexible! (migration, checkpointing, etc.) (+)
  - significant performance penatly (–)
- paravirtual drivers
  - needs extra effort to build the interface to the native driver (–)
  - scalable, as the interface to both VMs and the hardware is tailored to each driver class (+)
- IOV: hardware multiplexing
  - native drivers everywhere (+)
  - near-native performance (+)
  - intrusive in terms of hypervisor support (–)
  - inflexible (-)

I/O Virtualization (IOV)

- direct data paths (near-native performance in terms of I/O)

I/O Virtualization (IOV)

- direct data paths (near-native performance in terms of I/O)
- flexibility, migration, scalability (?)

I/O Virtualization (IOV)

- direct data paths (near-native performance in terms of I/O)
- flexibility, migration, scalability (?)

In the era of multi/many–cores:

- VM containers will host a great number of VMs
- are IOV adapters ready ?

Characterizing I/O devices

Block Devices

- batching requests at block level – difficult to virtualize due to hardware characteristics (e.g. single set of spindles will serialize access – needs a special interface to batch requests and then submit them)

Network devices

- batching transmision at packet level – multiple queues (TX/RX)

Accelerator Hardware

- GPUs
- FPGAs

both virtualizable depending on workload/hardware characteristics – not yet implemented

Overview

High-performance Interconnects – Basic concepts

Endpoints

virtualized instance of a device – logical source or destination of all communication

Regions

sets of memory segments that contain virtually contiguous memory areas (allocated by the application)

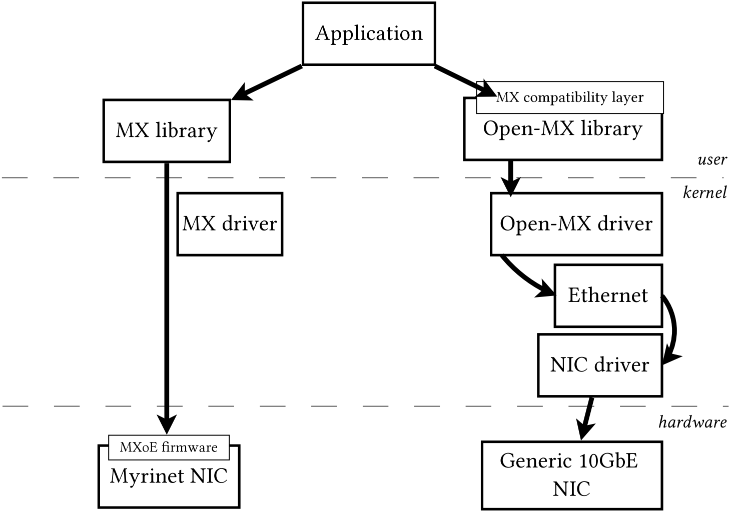
Events

- user-level
- kernel-level

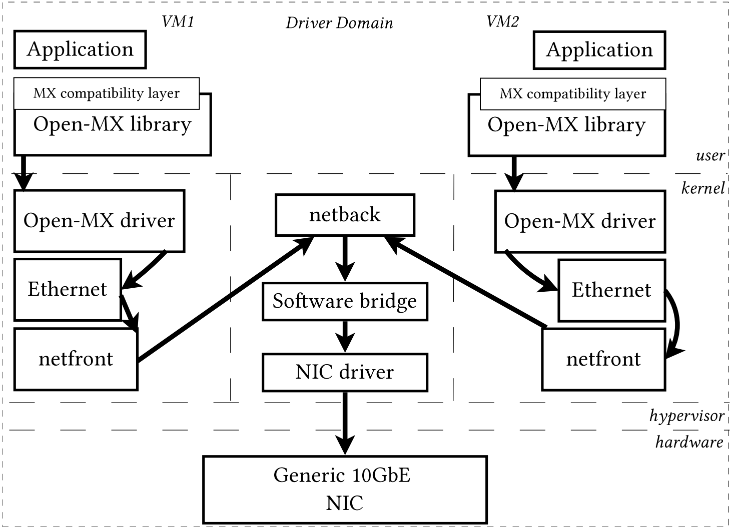
scalable method of communication between user–space and the hardware

National Technical University of Athens CSLab

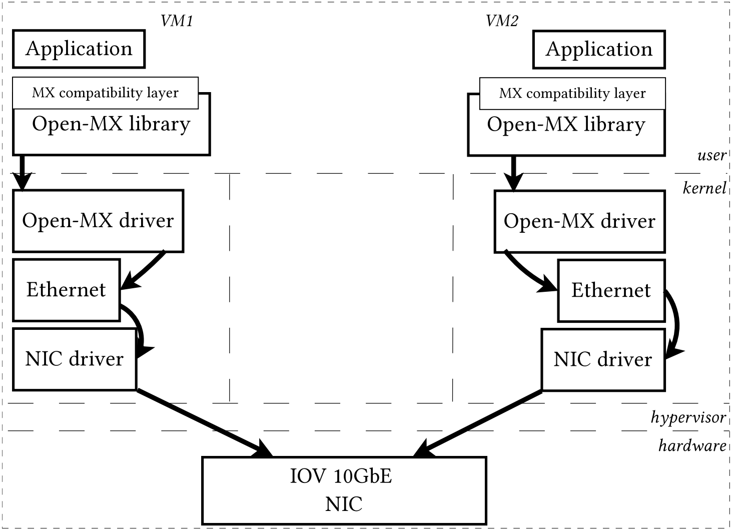
Open-MX software stack (user-level vs. traditional communication)



Open-MX deployed in a bridged setup

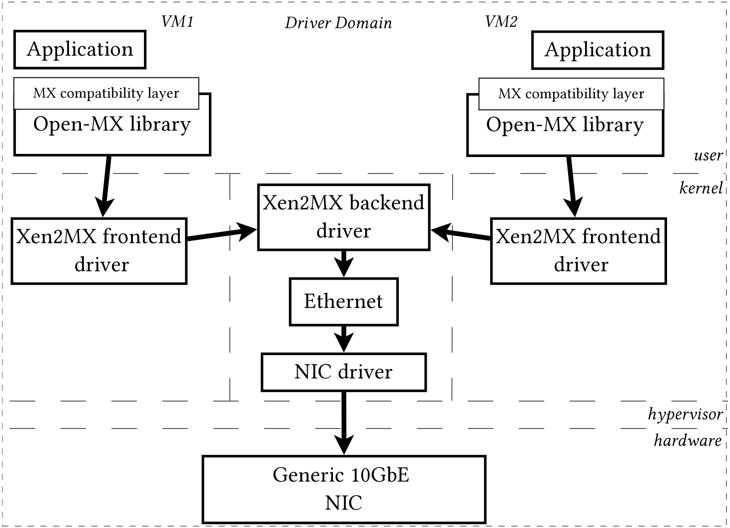


Open-MX deployed in an IOV setup



Overview

Xen2MX architecture



Xen2MX – architecture details

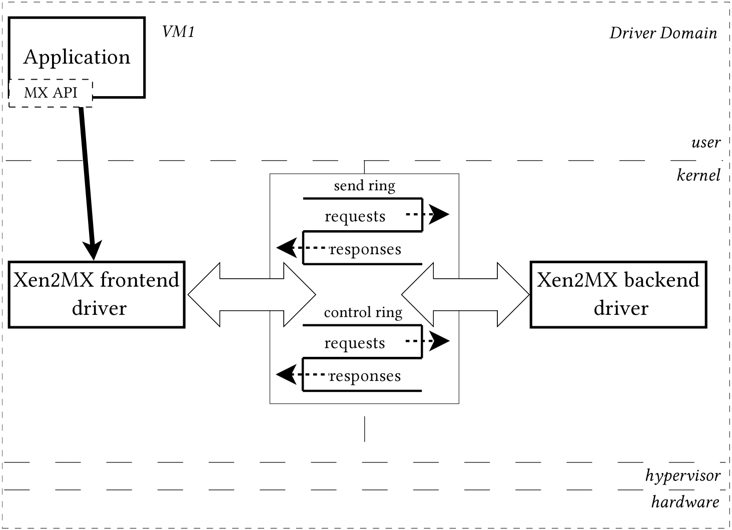
Frontend–Backend communication

- consumer-producer scheme (soft-interrupts/polling)
- cyclic rings
- anticipatory handlers

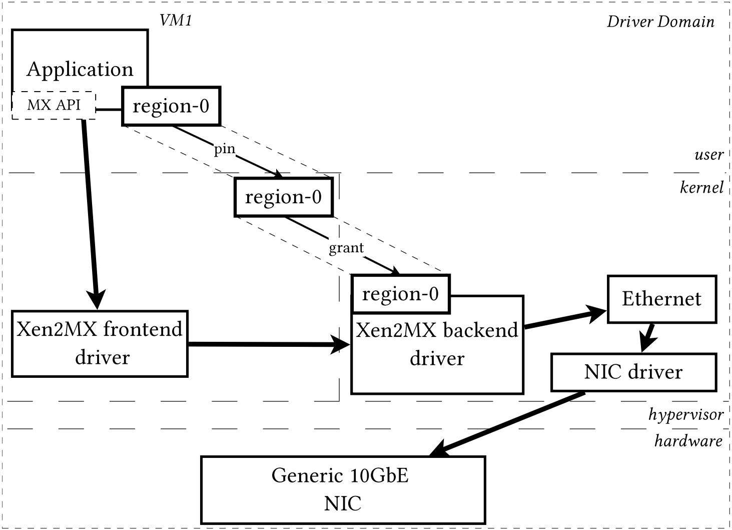
Data Exchange (inter–/intra–node)

- copies (`SMALL` messages)
- send & receive queues (`MEDIUM` messages)
- regions (`LARGE` messages)
- grants (proactive):
  - pre-grant relevant memory space (both send and receive)
  - re-use grants

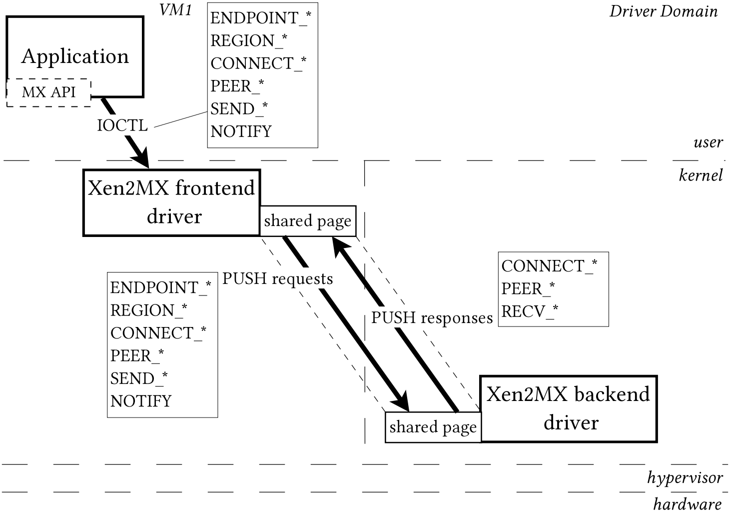
Xen2MX message exchange



Xen2MX regions



Xen2MX API


VM1
Driver Domain
Application
MX API
IOCTL
ENDPOINT_*
REGION_*
CONNECT_*
PEER_*
SEND_*
NOTIFY
user
kernel
Xen2MX frontend driver
shared page
PUSH requests
PUSH responses
ENDPOINT_*
REGION_*
CONNECT_*
PEER_*
SEND_*
NOTIFY
CONNECT_*
PEER_*
RECV_*
Xen2MX backend driver
shared page
hypervisor
hardware

Xen2MX performance results

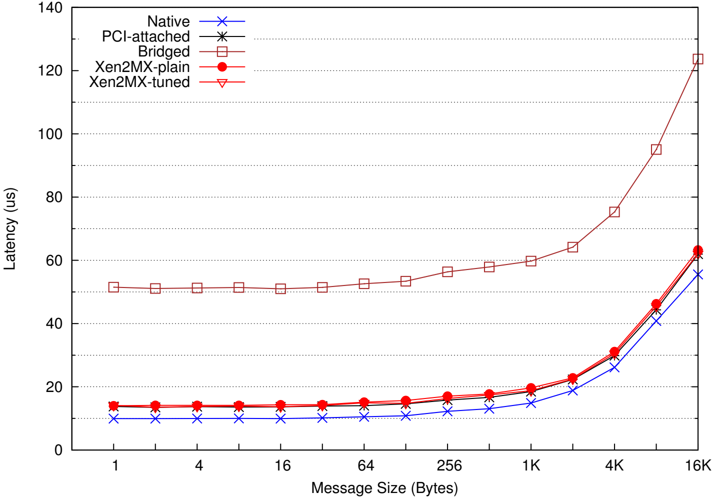
Testbed

- 2x {Intel Xeon @2.4Ghz, Intel 5500, 48GB memory, Generic 10GbE}
- Xen 4.2, Open-MX 1.5.2, Debian GNU/Linux (kernel version 3.4.0)
- generic microbenchmark: `mx_pingpong`
  - $\leq$ 64b: copying
  - $\leq$ 32KB: send & receive queues
  - $\geq$ 64KB: rendez-vous semantics

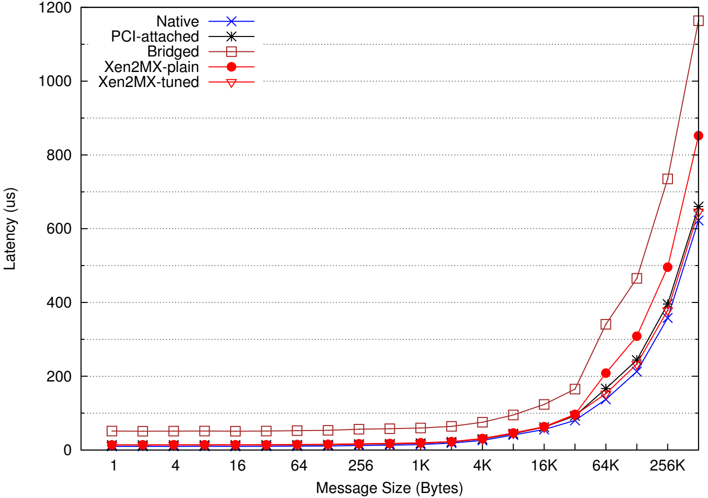
Cases:

- Native (no hypervisor)
- PCI-attached (IOV-equivalent)
- Bridged (Generic case)
- Xen2MX (Plain, Tuned)

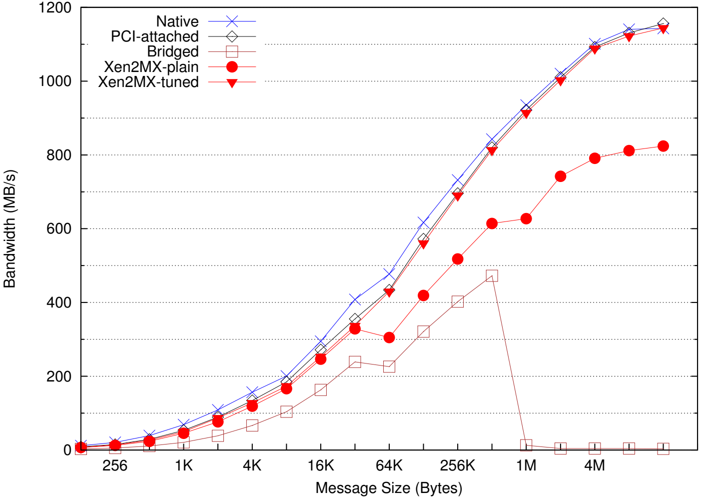
Xen2MX performance results – latency up to 16K



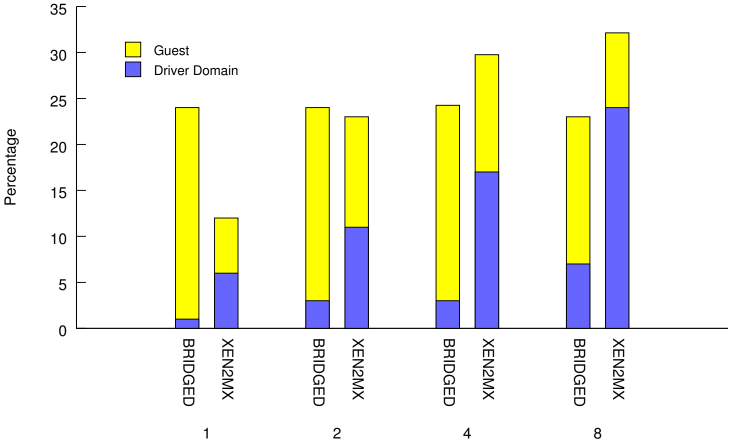
Xen2MX performance results – latency



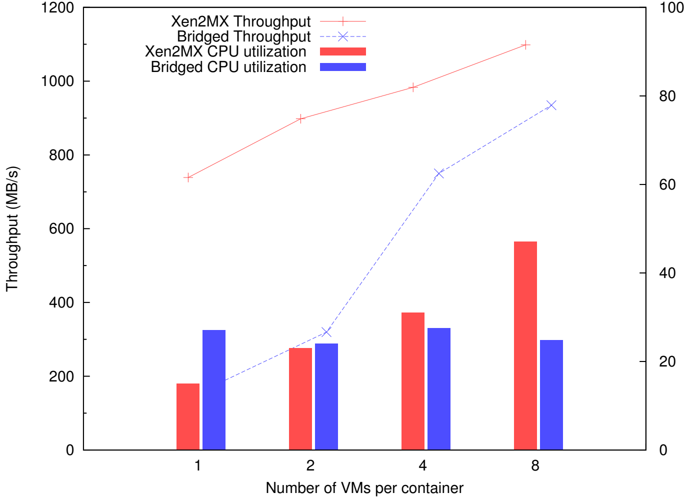
Xen2MX performance results – throughput



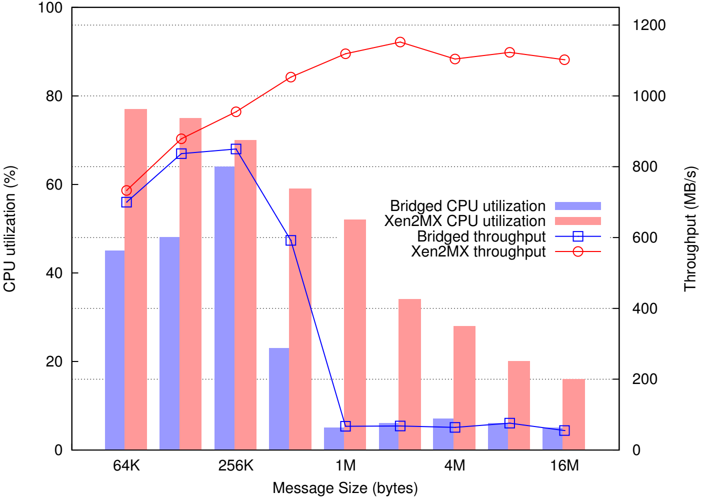
Xen2MX performance results – 512K messages



Xen2MX performance results – 256K messages



Xen2MX performance results – scale up to 40 VMs



Overview

Summary

I/O options in VM environments

- emulated (when there is no other option)
- split driver model (generic, scalable)
- IOV (expensive but provides near-native performance)

Xen2MX

is a software port of the MX protocol to the Xen split driver model.

Xen2MX benefits from:

- all Open-MX features (MX binary compatibility, MXoE wire compatibility)
- low-latency communication (almost as low as the IOV case)

Get Xen2MX!

`https://github.com/ananos/xen2mx`

8th Workshop on Virtualization in High-performance Cloud Computing

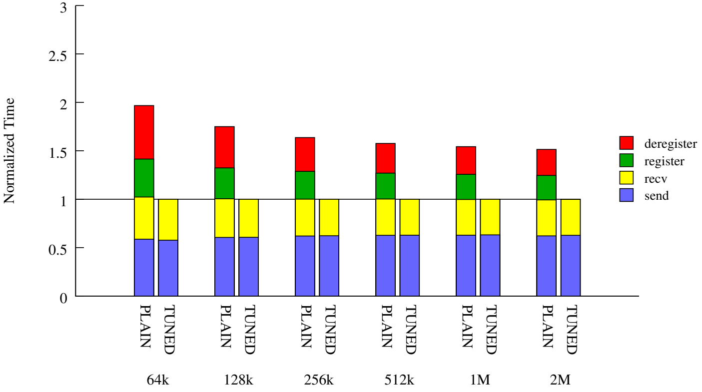
The Benefits and Challenges of vHPC and Cloud Computing
Josh Simons, Office of the CTO, VMware
Integration of High-Performance Computing into a VCL Cloud
Patrick Dreher, MIT
"Archipelago: Unified Storage over Commodity Hardware using RADOS"
Vangelis Koukis, Technical Lead, okeanos cloud at GRNET
The Evolution of the ARM Architecture Towards Big Data and the Data-Centre.
John Goodacre, Director, Technology and Systems, ARM Processor Division
Maximizing Performance with Cloud-Virtualized Dataflow Engine co-Processors
Jacob Bower, VP of Application Engineering, Maxeler Technologies.
Closing Discussion

Thanks!

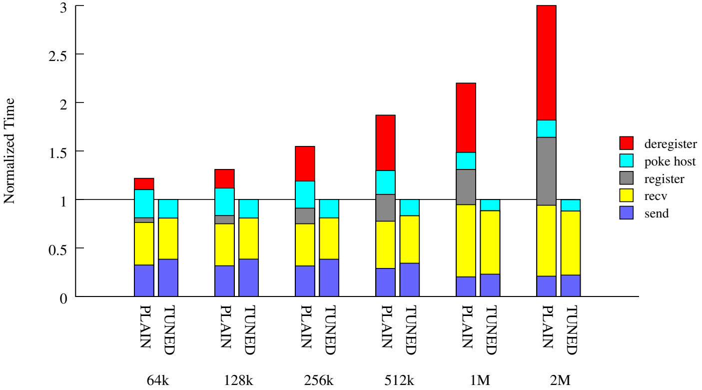
Questions?

Backup

Xen2MX performance results – Host CPU utilization



Xen2MX performance results – Guest CPU utilization



Xen2MX steps – I

Control VM *(aka driver domain): has direct access to the hardware and controls all VM operations.*

Xen

Generic 10GbE NIC

Smart 10GbE NIC

Hardware

Guest VM: *runs the user application.*

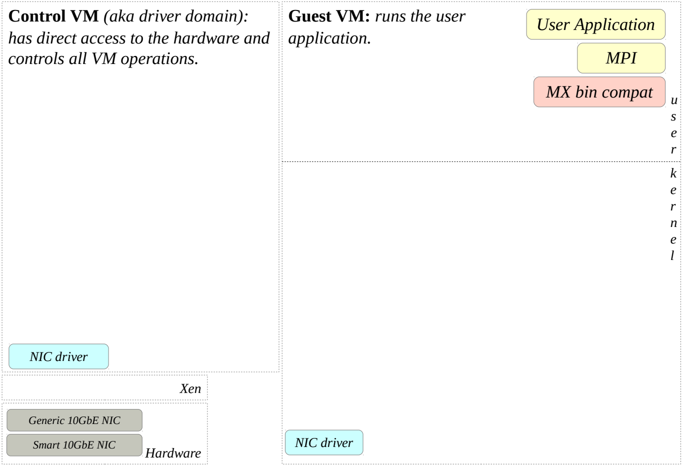
User Application

MPI

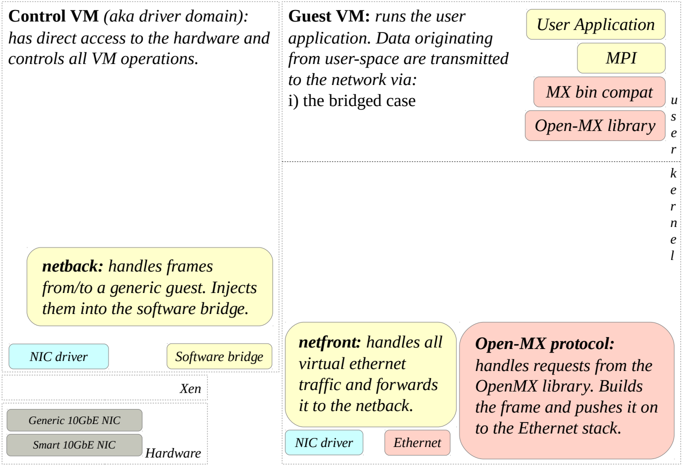
user

kernel

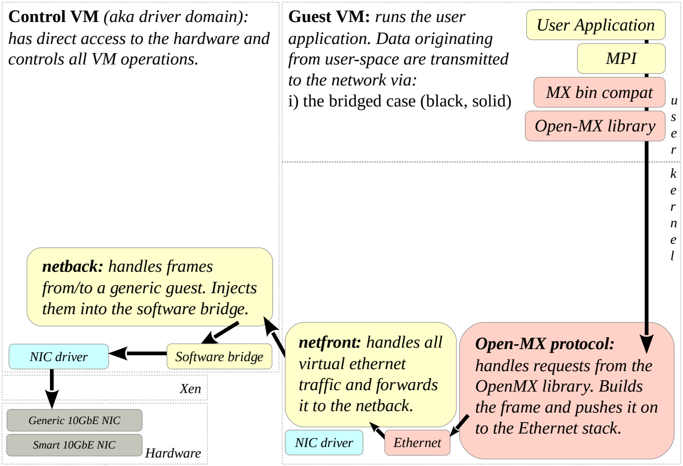
Xen2MX steps – II



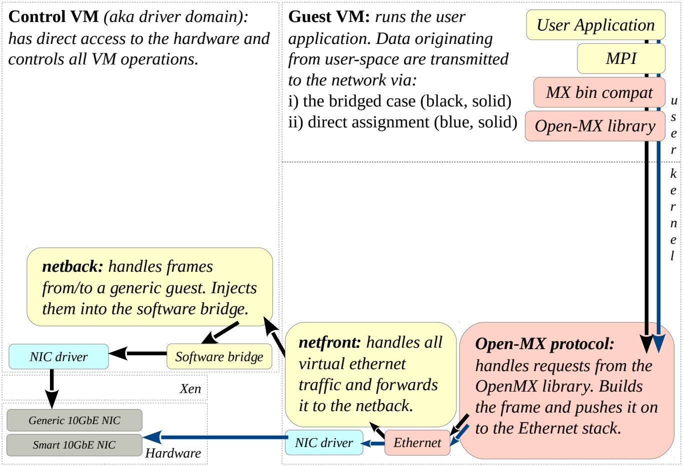
Xen2MX steps – III



Xen2MX steps – IV (Bridged)



Xen2MX steps – V (IOV)



Xen2MX steps – VI (Xen2MX)