MIKELANGELO

D3.1
The First Super KVM - Fast virtual I/O hypervisor

Workpackage	3	Hypervisor Implementation	
Author(s)	Yossi Kuperman		IBM
	Joel Nider		IBM
	Shiqing Fan		Huawei
	Holm Rauchfuss		Huawie
Reviewer	Michael Gienger		HLRS
Reviewer	Nadav Har'el, Benoît Canet		ScyllaDB
Dissemination Level	PU		

Date	Author	Comments	Version	Status
2015-12-10	Joel Nider	Initial draft	V0.0	Draft
2015-12-13	Joel Nider	Document ready for review	V0.1	Review
2015-12-20	Yossi Kuperman	Applied reviewers' comments	V0.2	Review
2015-12-28	Yossi Kuperman	Final	V1.0	Final

Executive Summary

sKVM, the *super* KVM, is an enhanced version of the popular Kernel-based Virtual Machine (KVM) hypervisor. It improves the performance of virtual I/O devices, such as disks and network interfaces, by changing the underlying threading model in the virtual device backend (vhost). sKVM furthermore provides a new type of virtual I/O device: virtualized RDMA (Remote Direct Memory Access) device. This device abstracts the behaviour of the physical RDMA device and offers the flexibility of the virtualized infrastructure by reducing the performance overhead of existing solutions.

sKVM has been designed specifically with backwards compatibility with the existing KVM interfaces and tools in mind in order to facilitate seamless transition between systems. There are no specific requirements for the guest Virtual Machines (VM), allowing complete reuse of existing resources, but with the new and improved I/O management features offering immediate improvements to end users.

This report documents the improvements made to the Linux kernel, and how to correctly use these new features for improved server performance. Much of the work has been done on the definition of a stable Application Programming Interface that is going to drive the development of the hypervisor components. Preliminary results already show the initial improvement of performance due to the work performed in the first project year. On the other hand, the same results demonstrate the need for those components of the overall architecture that are planned for the next stage of the project.

Acknowledgement

The work described in this document has been conducted within the Research & Innovation action MIKELANGELO (project no. 645402), started in January 2015, and co-funded by the European Commission under the Information and Communication Technologies (ICT) theme of the H2020 framework programme (H2020-ICT-07-2014: Advanced Cloud Infrastructures and Services)

Table of contents

Table of Figures

List of Abbreviations

API	Application Programming Interface
CPU	Central Processing Unit
NIC	Network Interface Controller
TCP	Transmission Control Protocol
HPC	High-Performance-Computing
I/O	Input / Output
RAM	Random Access Memory
RAM-disk	In Memory filesystem
RDMA	Remote Direct Memory Access
RoCe	RDMA over Converged Ethernet
VM	Virtual Machine
QEMU	Quick Emulator
KVM	Kernel-based Virtual Machine
sKVM	super KVM
IOcm	I/O Core Manager
vRDMA	virtual RDMA
OFED	Open Fabrics Enterprise Distribution

1 Introduction

This document describes our contributions to sKVM: I/O core manager (IOcm) and lightweight RDMA para-virtualization (vRDMA).

The sKVM architecture has first been described in detail in report D2.13[1]. The following figure shows all the components that constitute the highly optimised new kernel-based hypervisor: IOcm, vRDMA and SCAM (Side Channel Attack Monitor and Mitigation). IOcm and vRDMA are described next.

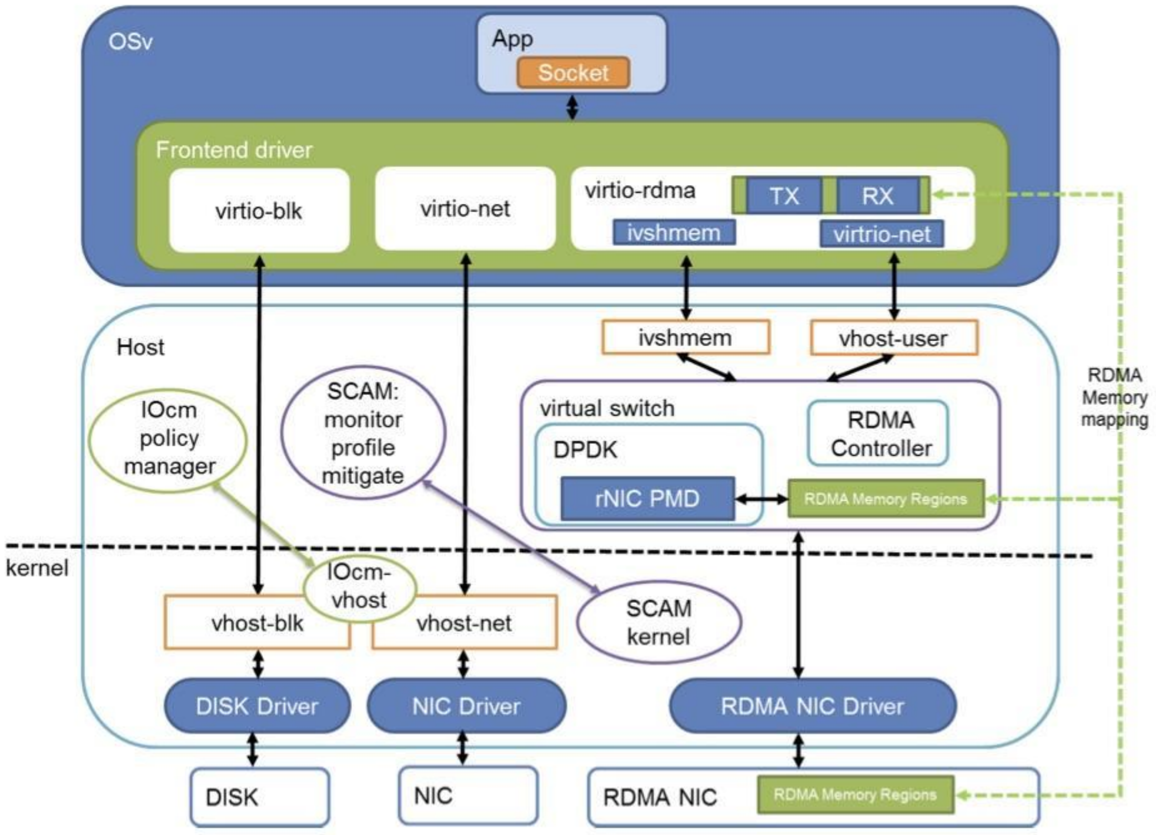

Figure 1: The high-level architecture diagram of sKVM

The KVM hypervisor depends on the virtio[2] protocol to implement virtual I/O devices for guests. The backend of the virtio protocol is implemented in a Linux kernel module called vhost. IOcm extends the vhost module to provide more control over the allocation of resources for handling virtual I/O.

A lightweight RDMA para-virtualization solution (a.k.a. vRDMA) has been designed and the first prototype has been implemented in the first year of the MIKELANGELO project. The design contains a frontend driver in the guest that supports for both, RDMA and TCP protocols for the application, and a backend driver that manipulates the real RDMA operation on the physical RDMA-capable device. An additional feature of the design is efficient inter-VM communication. Inter-VM communication for co-located VMs takes place over a shared memory protocol removing the host from the critical path. The objective in this work is to develop techniques and mechanisms to accelerate virtual I/O using RDMA hardware, and improve scalability for multiple virtual machines running on a multi-core host. By accelerating the virtual I/O, we reduce the performance overhead incurred by virtualization, thereby making Cloud and HPC more efficient.

2 IOcm

IOcm builds on ELVIS[3] technology, which introduced efficiency improvements to the I/O subsystem of the hypervisor. By using a shared I/O processing thread, the hypervisor is then able to take control of its own decisions regarding the scheduling policies of I/O processing. By relieving the general Linux thread scheduler, and transferring the responsibility to the hypervisor, the system gained a higher level of control over I/O traffic, and less wasted overhead of thread context switching. The shared I/O processing thread allows the hypervisor to control processing per virtual device very precisely, allowing the hypervisor to make rapid scheduling decisions in response to a changing environment. This alleviates thread starvation, and threads that held the CPU despite not having any active I/O requests.

IOcm is a mechanism to control the aforementioned shared I/O threads. It enables low-level functionality such as creating and destroying I/O threads (vhost), and migrating devices between I/O threads. At a higher level, the main component of IOcm is the monitor, which makes decisions about resource allocation to ensure maximum I/O performance. The monitor periodically reads statistics such as the throughput, and uses these statistics to determine the optimum configuration of the I/O subsystem of the hypervisor (vhost thread).

The vhost module is controlled from user-space through an API implemented using sysfs[4]. Sysfs is a common mechanism in the Linux kernel for providing information and control points to user space. Specific data is exposed through a set of virtual files which can be used to understand the state of vhost, and modify its behaviour accordingly. The files exposed through sysfs appear in the file system as regular files owned by 'root', and are subject to the same treatment as other files (ownership, access controls, etc).

An application in user space can communicate with the vhost module by reading and writing these sysfs files. Together, these files constitute the vhost IOcm interface (API), which is described in detail in the following sections.

2.1 Linux Kernel with IOcm

To get a copy of the IOcm code, clone this git:

```
ssh://git@gitlab.xlab.si:13022/mikelangelo/linux-3.18.y.git
```

All the changes are applied to existing modules. To use the modified kernel compile it as usual. Then follow the API specification described next to use the IOcm capabilities.

2.2 IOcm API

The API is centered around 3 main objects: virtual queue, device, and worker. For completeness, we provide all the available APIs as part of the changes made to the Kernel. Some of which are statistical measurements for experimenting and evaluating new features and are subject to change.

Virtual Queue The virtual queue represents a unidirectional stream of data between the virtio frontend and virtio backend (or vice versa). Devices that are slaves (such as disks) may only have a single queue, since data is transferred as request-reply pairs. Other devices (such as Network Interface Controllers) may have multiple queues, since the incoming data flow is asynchronous in nature (thus, a separate queue for transmitting - tx and receiving - rx). Each virtual queue is owned by a single logical device, and cannot be transferred to another device.

Device The device object represents the backend of a virtio device. It owns one or more virtual queues, which handle all of the data transfer on behalf of the device. The device is handled by a single worker at any given time, and can be transferred to another worker upon request.

Worker The worker represents a thread of execution that handles I/O requests on behalf of a particular set of (one or more) devices. There is one worker per I/O core, and there may be multiple I/O cores in a system. The worker owns one or more devices (which in turn contains one or more queues). Worker will be destroyed if the I/O core is to be repurposed to run more VMs, at which point all of the devices handled by this worker must be migrated to a different worker. Workers are destroyed when the number of I/O cores is decreased, or when changing mode from non-shared worker state (traditional vhost processing) to shared worker state.

There are 3 main functions in the IOcm kernel module:

Enable IOcm IOcm can be enabled and disabled at run time. Disable IOcm returns hypervisor to normal KVM functionality--vhost thread per device.

Allocate an I/O core To maximize the system performance, cores can be allocated for I/O or for vCPUs as necessary. Vhost threads are created or destroyed as necessary to keep the 1 to 1 relationship to I/O cores.

Migrate device Devices can be migrated between worker instances (I/O cores) for load balancing.

Workers (*/sys/class/vhost/worker*)

The directory containing all the statistics and controls for IO workers:

File	Description
create	Writing a CPU mask creates a new worker with affinity to the requested CPUmask, Write 0 to specify no affinity. Reading returns the array of newly created workers id since the last time this file was read (upto 32 workers).
default	Writing a worker ID set the worker with the same ID as the designated worker new devices are assigned to. The worker must be unlocked. Reading returns the current value.
remove	Writing a worker ID removes a worker with the same ID. The worker is removed only if it is locked and has no queues assigned to it.
worker ID	Each I/O worker has a directory containing all its statistics and controls. This accesses the same information available through Workers (/sys/class/vhost/w.[XXX]).

Cycles (/sys/class/vhost/cycles)

Returns the current value of the timestamp counter. The counter is used to calculate the throughput (i.e. how many bytes per second passed through a device).

Epoch (/sys/class/vhost/epoch)

A way to quantify time. Writing 0 advances the epoch. Reading returns the current epoch.

Devices (/sys/class/vhost/d.[XX])

Contains all the statistics and controls for I/O devices:

File	Description
owner	Reading returns the PID of the owner thread (a.k.a the VM).

File	Description
vq list	Reading returns a newline separated list of ids of virtual queues owned by the device.
worker	Writing a I/O worker ID transfers a device from its current worker to the worker with the written ID. The receiving worker cannot be a locked worker. Reading returns the current value.

Virtual Queues (/sys/class/vhost/vq.[XX].[YY])

The directory containing all the statistics and controls for virtual queues contained in the virtual I/O devices. XX refers to the device ID that owns the virtual queue and YY is the queue index starting from 0 for that device.

File	Description
can_poll	Reading returns 1 if the queue can be polled, 0 otherwise.
dev	The ID of the device that currently owns this queue. Reading returns the number of bytes handled by this queue in the last poll/notif. Must be updated by the concrete vhost implementations (i.e. vhost-net).
ksoftirq_occurrences	Reading returns the number of times a measurement of I/O activity time had a softirq interrupt occurring during it.
ksoftirqs	Reading returns the number of ksoftirq interrupts that occurred during the measuring of I/O activity inside the I/O thread of this specific virtual queue.
last_poll_cycles	Reading returns the end of the last polling in cycles.
max_processed_data_limit	Writing sets the maximum amount of bytes to be processed in a single service cycle of virtual queue (disable = -1). Reading returns the

File	Description
	current value.
max_stuck_cycles	Writing sets the number of need to elapse without service to consider the queue stuck (disable = -1). Reading returns the current value.
max_stuck_pending_items	Writing sets the maximum number of pending items in the queue to consider the queue stuck (disable = -1). Reading returns the current value
min_poll_rate	Writing sets the minimum rate in which a polled queue can be polled (disable = 0). Reading returns the current value.
min_processed_data_limit	Writing sets the minimum amount of bytes to be processed in a single service cycle of virtual queue (disable = 0). Reading returns the current value.
notif_bytes	Reading returns the number of bytes sent/received by works in notif mode
notif_cycles	Reading returns the number of cycles spent handling works in notif mode
notif_limited	Reading returns the number of times the queue was limited by netweight in notif mode.
notif_wait	Reading returns the number of cycles elapsed between between work arrival and handling in notif mode.
notif_works	Reading returns the number of works in notif mode.
poll	Writing starts/stops polling of the virtual queue. Reading returns the current value.

File	**Description**
poll_bytes	Reading returns the number of bytes sent/received by kicks in poll mode.
poll_coalesced	Reading returns the number of times this queue was coalesced.
poll_cycles	Reading returns the number of cycles spent handling kicks in poll mode.
poll_empty	Reading returns the number of times the queue was empty during poll.
poll_empty_cycles	Reading returns the number of cycles elapsed while the queue was empty.
poll_limited	Reading returns the number of times the queue was limited during poll kicks.
poll_pending_cycles	Reading returns the total amount of cycles all the work items in the ring buffer waited for service.
poll_wait	Reading returns the number of cycles elapsed between poll kicks.
ring_full	Reading returns the number of times the ring was full.
stuck_cycles	Reading returns the number of amount of cycles the queue was stuck.
stuck_time	Reading returns the number of times this queue was stuck and limited by other queues.

Workers (/sys/class/vhost/w.[XXX])

File	**Description**
cpu	Reading returns the core affinity of the worker.
cycles	Reading returns the cycles spent in the worker, excluding cycles performing queue work.
dev_list	Reading returns a newline separated list of IDs of devices assigned to this worker.
empty_polls	Reading returns the number of times there were no queues to poll and the polling queue was not empty.
empty_works	Reading returns the number of times there were no works in the queue – ignoring poll kicks.
ksoftirq_occurrences	Reading returns the number of times a measurement of I/O activity time had a softirq interrupt occur during processing.
ksoftirqs	Reading returns the number of ksoftirq interrupts that occurred during the measuring of of I/O activity inside the I/O thread of this specific worker.
ksoftirq_time	The time (ns) that softirq thread consumed while the worker processed its work (this measurement is very coarse-grained and as such is unreliable).
ksoftirq_time_clock_t	Reading returns the time (in clock ticks) that the softirq process consumed while the worker processed its work (this measurement is very coarse-grained and not accurate).
locked	Writing 1 sets the worker as locked, when a worker is locked it cannot be assigned queues. Reading returns 1 if device is locked, 0 otherwise. Note: once a worker has been locked it cannot be unlocked; only

File	Description
	removed.
loops	Reading returns the number of loops performed by the worker.
max_disabled_soft_interrupts_cycles	The maximum number of cycles a worker is allowed to turn off soft interrupts while servicing poll queues.
mm_switches	Reading returns the number of times the memory context (mm) was switched.
noqueue_works	Reading returns the number of pending works.
notif_cycles	Reading returns the cycles spent in the worker handling kicks in notification mode (i.e. not polling mode).
pending_works	Reading returns the number of works, which have no queue related to them (e.g. vhost-net rx).
pid	Reading returns the process ID of the worker.
poll_cycles	Reading returns the cycles spent in the worker handling kicks in poll mode.
stuck_works	Reading returns the number of times a queue was stuck or limited, hence could not make any progress.
total_softirqs	Reading returns the total number of softirq interrupts on this CPU since boot.
total_work_cycles	Reading returns the cycles spent in the worker handling work in any mode.

File	Description
wait	Reading returns the number of cycles the worker thread was not running after schedule.
work_list_max_stuck_cycles	The maximum number of cycles need to elapse to consider the work list stuck (disabled = -1).

2.3 Evaluation

Test-bed

Our test system is comprised of two physical machines: a load generator and a machine that hosts the Virtual Machines (VMs). Both machines are identical and of type IBM System x3550 M4, equipped with two 8-cores sockets of Intel Xeon E5-2660 CPU running at 2.2 GHz, 56GB of memory and two Intel x520 dual port 10Gbps NICs. All machines run Ubuntu 12.04 with Linux 3.9 (guests, host, and load generator). The host’s hypervisor is KVM with QEMU 1.3.0. Hyperthreading and all power management features are disabled in the BIOS.

The machines are connected in a back-to-back fashion as depicted in Figure 2.

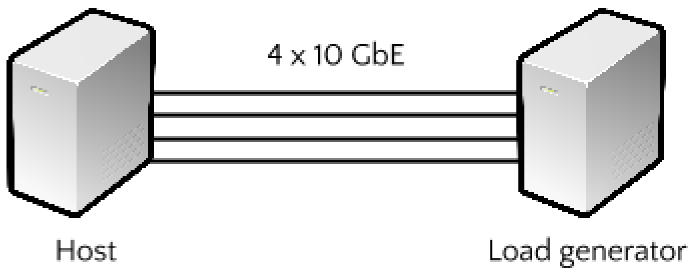

Figure 2: Test system setup

Methodology

Each experiment is executed 5 times, 60 seconds each. We made sure the variance across the results (both performance and CPU utilization) is negligible, and present their average. Benchmark parameters were meticulously chosen in order to saturate the vCPU of each VM.

Configurations

- **Baseline** We use KVM virtio as the state-of-practice representative of paravirtualization. We denote it as the baseline configuration

- **elvis-X** Our modified version of vhost, with a different number of dedicated I/O cores (1-7), denoted by X.

With all configurations, we set the number of VMs to be 12[1] (overcommit) throughout all the benchmarks, utilizing only one 8-core socket. Each VM is configured with 1 vCPU, 2GB of memory and a virtual network interface. All four physical ports are in use and assigned evenly between VMs. The NICs are connected to the VMs using the standard Linux Bridge.

With the "elvis-X" configuration, we vary the number of I/O cores from 1 to 7 (constrained by our 8-core socket.) at the expense of available VM cores. Given a number of I/O cores, the VMs are assigned in a cyclic fashion to the remaining cores. For the "baseline" setup, there is no affinity between activities and cores, namely, interrupts of the physical I/O devices, I/O threads (vhost), and vCPUs.

Preliminary results

Netperf[5] Our first experiment evaluates a throughput-oriented application. We use the Netperf TCP stream for this purpose, which measures network performance by maximizing the amount of data sent over a single TCP connection, simulating an I/O-intensive workload. We vary the message size between 64 and 16384 bytes.

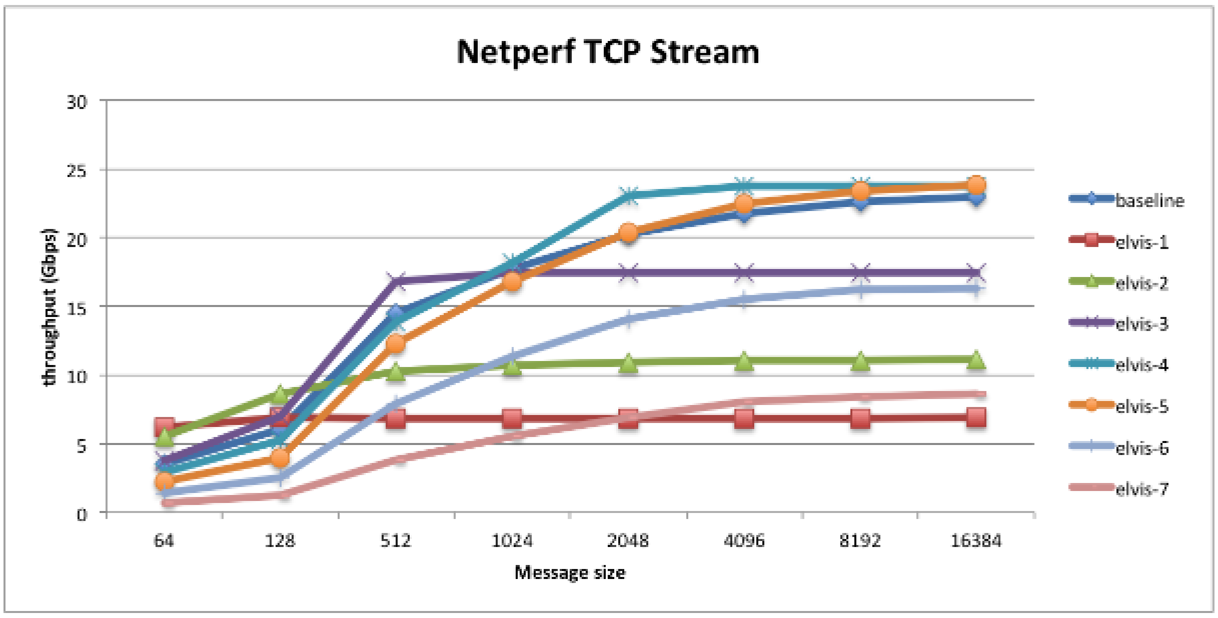

Figure 3: Performance comparison of netperf TCP stream between baseline and elvis-X

[1] We are mostly interested in the first four elvis-X configurations. To achieve balanced results, each I/O core is assigned equal number of VMs. 12 is evenly divided by 1, 2, 3 and 4.

With elvis-X configurations, each additional I/O core comes at the expense of the cores that are available for running VMs. For example, elvis-7 dedicates 7 I/O cores and only 1 core is shared among the 12 VMs. Naturally, an I/O core has a limit to the amount of traffic it can handle in a given period. For elvis-X, we can see the throughput curves become flatter at a certain point as message size increases. In elvis-1, the I/O core is saturated with the smallest message size, while for elvis-2 both I/O cores reached their maximum capacity with message size 512. Additionally, elvis-6 and elvis-7 exhibits poor performance compared to the other configurations. This is due to the system being underutilized, 6/7 I/O cores are mostly idle and only 2/1 cores are available for all the VMs to execute.

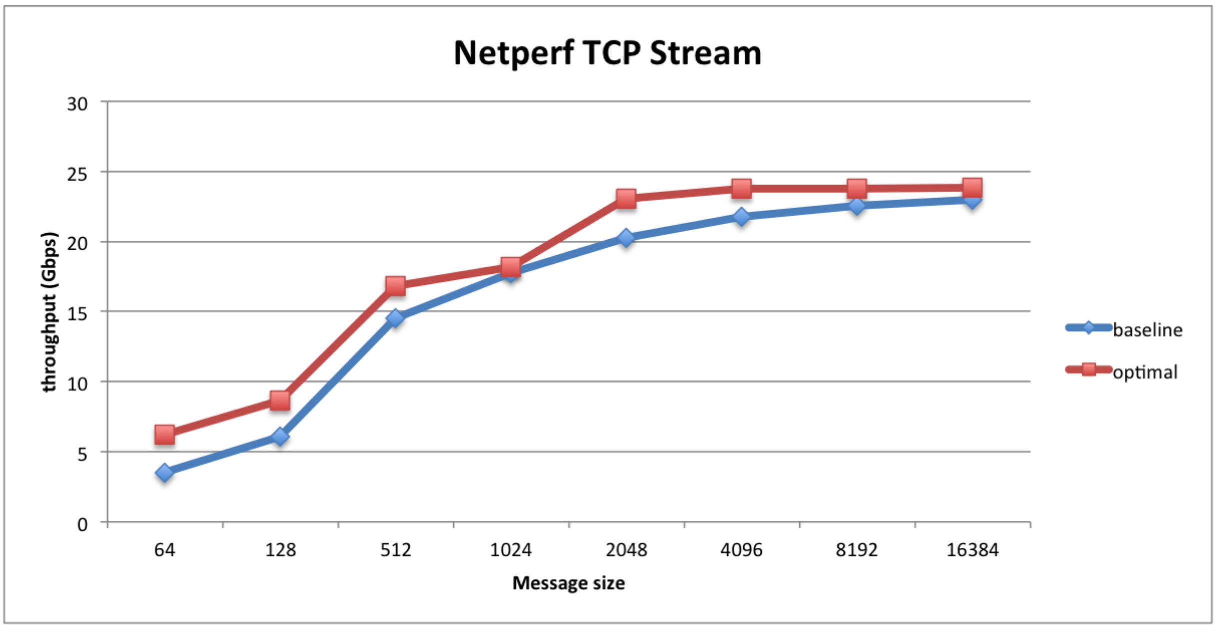

Figure 4: Performance comparison of netperf TCP stream between the baseline and the best of elvis-X (per message size)

The above figure presents the baseline result alongside the best of elvis-X configurations, depicted as "optimal". Going forward, the IOcm policy manager will be responsible to sample the system-state frequently enough to determine which of the various elvis-X configurations is the best at that given time and to act accordingly.

Apache[6] To evaluate the performance on a real application, we use Apache HTTP Server. We drive it using the ApacheBench[7] (also called "ab") which is distributed with Apache. It assesses the number of concurrent requests per second that the web server is capable of handling. We use 16 concurrent requests per VM for different file sizes, ranging from 64 bytes to 1 MB. The results are shown at the following figure.

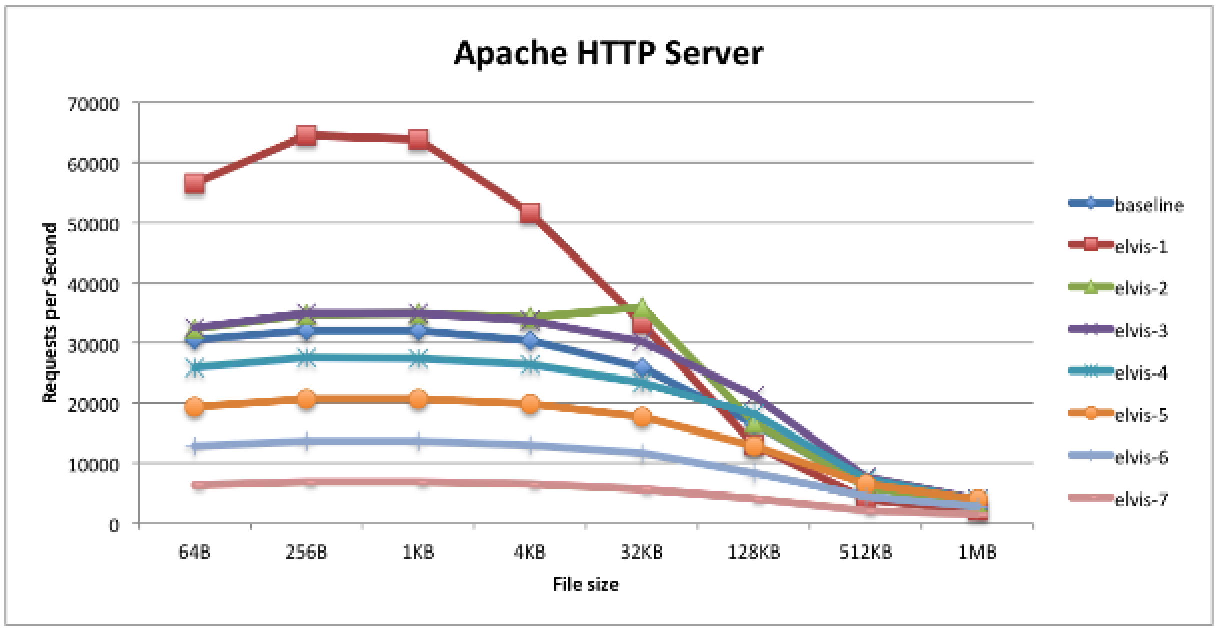

Figure 5: Performance comparison of Apache HTTP server between the baseline and elvis-X

elvis-1 clearly outperforms the baseline for smaller requests, and converges for larger files. According to the ELVIS paper[3], ELVIS improves the latency over the baseline configuration. Latency is more dominant when requesting small files and it becomes more stream oriented when large files are serviced. This explains the gap between elvis-1 and baseline configurations.

Two unanticipated outcomes can be observed from the above figure. The first is that elvis-2 falls behind elvis-1 considerably, almost by half. Second, we would expect the run with the smallest request size to yield the best performance, in terms of requests per second, but we can see a slight improvement for 256 and 1024 bytes. These unexpected results are not yet fully understood and require further investigation.

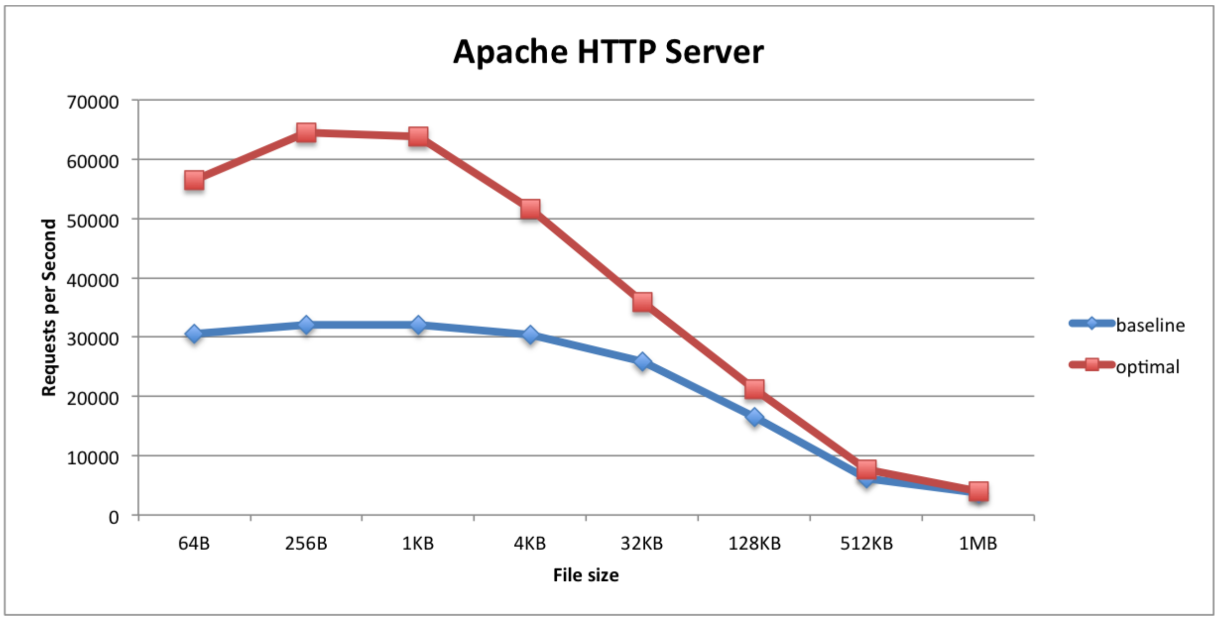

Figure 6: Comparing the performance of Apache HTTP server between the baseline and the best of elvis-X (per file size)

Similar to netperf, we present the baseline result for Apache compared to the optimal. This again shows the need for dynamic monitoring and management of cores reserved for I/O operations based on the current workload.

3 vRDMA

3.1 Overview

We have presented three prototypes for the lightweight RDMA para-virtualization solution in the M8 deliverable D2.13[1] (The first sKVM hypervisor architecture) and M9 deliverable D2.16[8] (The First OSv Guest Operating System MIKELANGELO Architecture). In this section, we will give an overview of the entire design and discuss briefly the basic components on the host. The implementation, integration and achievements on this design are described in the M12 deliverable D4.1[9] (The First Report on I/O Aspects).

Figure 7 shows an abstraction of these prototypes. The frontend driver, i.e. the virtio-rdma component on the guest, takes care of the communication requests of the guest application. It supports both RDMA and TCP protocols by using ivshmem [10] and the RDMA driver provided by Open Fabrics Enterprise Distribution (OFED). All the calls will be passed to the backend driver and processed on the host by DPDK rNIC PMD (Data Plane Development Kit RDMA Poll Mode Driver) [11] in prototype I. An Open VSwitch based virtual switch implementation is used for binding the vhost-user [12] port to the real physical port. The InfiniBand driver is supported and used based on the request from the guest and passed by DPDK rNIC PMD. In order to benefit from the kernel-bypass feature of the RDMA protocol, the RDMA memory regions are mapped on the host and shared with the guest and the RDMA device. The RDMA memory regions may be processed either by the host side PMD (prototype I), or by guest applications (Prototype II) or by the virtio-rdma frontend driver (Prototype III). The rest RDMA operations, which do not touch RDMA memory regions will be forwarded from the guest to the host, then corresponding operations will be performed by the RDMA driver on the host.

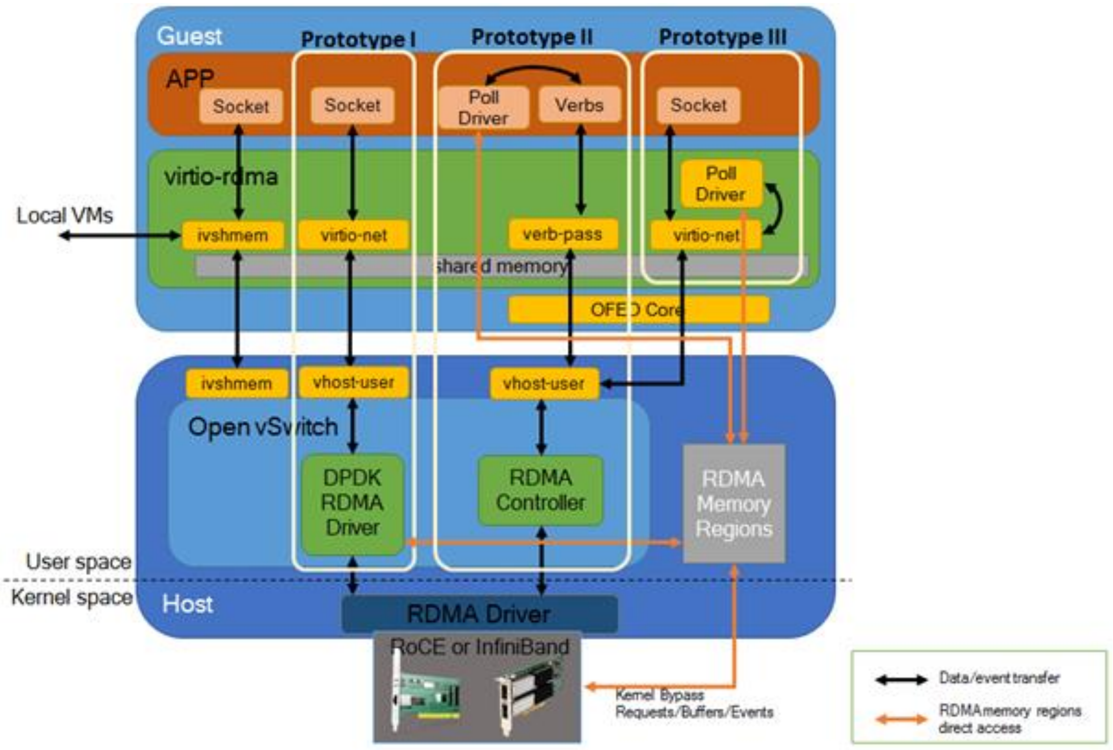

Figure 7: Architecture overview of the design prototypes

3.2 Usage Requirements

In order to support RDMA, virtualization technology must be supported by the CPU and enabled in the BIOS (e.g. VT-x for Intel CPUs and AMD-v for AMD CPUs). An Infiniband ConnectX-3 VPI adapter interconnected with proper cables and switches and its latest drivers must be installed.

There are several software packages that have to be installed and configured in order to use RDMA virtualization prototype I. These include:

- QEMU (at least version 2.2) for huge page and vhost-user support
- Open vSwitch version should be at least 2.4.0 for DPDK netdev support
- DPDK at least version 2.1.0 for InfiniBand Poll Mode Driver support
- libvirt requires at least version 1.2.19 for managing the VMs with easier configuration of the vhost-user device

3.3 External Packages on the Host

Open vSwitch

Open vSwitch[13] is a virtual switch that enables massive network automation through programmatic extension and supports standard management interfaces and protocols.

In design prototype I, Open vSwitch plays a key role in the entire system. It provides the virtual connections between the guest and the InfiniBand device on the host. With the DPDK support, the PMD is able to be started and manages the communication between the guest and the InfiniBand device.

It uses multi-queues to provide an approach that scales the network performance as the number of vCPUs increases, by allowing them to transfer packets through more than one virtqueue pair at a time. Multi-queue support removes these bottlenecks by allowing paralleled packet processing.

In a single queue virtio-net, the scale of the protocol stack in a guest is restricted, as the network performance does not scale as the number of vCPUs increases. Guests cannot transmit or retrieve packets in parallel, as virtio-net has only one TX and RX queue.

A guest running the virtio-net network driver will share a number of virtqueues with the QEMU process that hosts that guest. In this fashion, the QEMU process will receive the network traffic from the guest, and forward it to the host network. However, this implies that all guest traffic has to be handled by the QEMU process before it can be processed further by the host's network stack.

Virtio-net is used together with vhost-user in prototype I for the communication between guest and host.

vhost-user

Vhost provides in-kernel virtio device emulation and allows device emulation code to directly call into kernel subsystems instead of performing system calls from QEMU user space.

There are several device drivers that were developed based on vhost for emulating different device backend drivers in the host kernel, such as vhost-net, vhost-blk and vhost-scsi.

The vhost-user is a new implementation based on the kernel vhost, and it has been implemented in the latest versions of QEMU, Open VSwitch and DPDK. It works in the user space and uses kernel vhost to initialize the necessary resources that are shared between the processes in the user space. However, most of the communications are taking place in the user space. As the implementation on the host is in user space, using vhost-user for communication between the guest and the virtual switch will avoid additional switches between the kernel and user space. We use vhost-user to talk to the virtio-net in the guest.

ivshmem

Ivshmem is an inter-VM shared memory PCI device that facilitates fast zero-copy data and sharing among virtual machines (host-to-guest or guest-to-guest) by means of QEMUs ivshmem mechanism. It maps a shared memory object as a PCI device in the guest and supports interrupts between guests by communicating over a UNIX socket.

In this work, we will extend the functionalities of ivshmem to support the socket communication between guests. The component ivshmem in the host acts as the ivshmem server, which provides the shared memory service to the guest and performs the communication task with the host.

DPDK

DPDK provides a set of drivers and libraries for fast data processing. Starting with DPDK 2.0, a Mellanox Poll Mode Driver is published to support fast packet processing and low latency by providing kernel bypass for ConnectX-3 and ConnectX-4 devices.

This PMD is the key of prototype I, and it is integrated with Open vSwitch in order to bind with the virtual ports, which are connected to the guest through vhost-user.

3.4 Evaluation

In the M12 deliverable D4.1[9], we have presented the performance results based on a local testbed, which uses a HP ProDesk 600 with a 4-core Intel i7-4790 processor, and 16 GB RAM memory with the host operating system (OS)being Ubuntu 14.04 LTS. Guest OS is Ubuntu 14.04 server edition, which is configured with two vCPU (assigned with two isolated cores on the host) and 2GB of memory.

The initial performance test showed promising results, that the vRDMA prototype I implementation is about 25% of the bare metal performance. The evaluation on the bottleneck of vRDMA prototype I has shown that the PMD has to be optimized. This work is continuing along with the implementation of prototype II for the second project year.

4 Concluding Remarks

Both IOcm and vRDMA are geared towards improving virtual I/O performance. Both features are an integral part of MIKELANGELO, and form the basis of sKVM. The initial sKVM improvements are now implemented, and are entering the second phase of development. The preliminary results already demonstrate the improved performance in a controlled environment. In the coming year, we plan to evaluate and experiment our offerings with MIKELANGELO's use cases and focus on improving even further the relative efficiency of virtual I/O between KVM (our baseline) and sKVM (**KPI 2.1, O2)**.

This is an exciting stage of the project, as we will now start to see the benefits of the past year come to fruition. Future development towards the official MIKELANGELO technology stack is going to focus on functions deemed necessary for real-world scenarios exploitable beyond the scope of MIKELANGELO. These features, once they mature, will be excellent candidates to be integrated into mainstream Linux as part of KVM.

5 References and Applicable Documents

[1] MIKELANGELO Report D2.13 The first sKVM hypervisor architecture, https://www.mikelangelo-project.eu/deliverables/deliverable-d2-13/

[2] Russell, R. "virtio: Towards a De-Facto Standard For Virtual I/O Devices." 2012. ftp://ftp.os3.nl/people/nsijm/INR/Week%201/papers/32_virtio_Russel.pdf

[3] Har'El, Nadav et al. "Efficient and Scalable Paravirtual I/O System." *USENIX Annual Technical Conference* 26 Jun. 2013: 231-242.

[4] "sysfs - Wikipedia, the free encyclopedia." 2011. 17 Dec. 2015 https://en.wikipedia.org/wiki/Sysfs

[5] The Netperf Homepage. 2002. 17 Dec. 2015, http://www.netperf.org/

[6] The Apache HTTP Server Project. 2009. 17 Dec. 2015, https://httpd.apache.org/

[7] ab - Apache HTTP server benchmarking tool, https://httpd.apache.org/docs/2.2/programs/ab.html

[8] MIKELANGELO Report D2.16 The First OSv Guest Operating System MIKELANGELO Architecture, https://www.mikelangelo-project.eu/deliverables/deliverable-d2-16/

[9] MIKELANGELO Report D4.1 The First Report on I/O Aspects, https://www.mikelangelo-project.eu/deliverables/deliverable-d4-1/

[10] The C. Macdonell, X. Ke, A. W. Gordon, and P. Lu, “LOW-LATENCY, HIGH-BANDWIDTH USE CASES FOR NAHANNI / IVSHMEM,” Forum Am. Bar Assoc., pp. 1–24, 2011.

[11] I. Corporation, “Intel Data Plane Development Kit (Intel DPDK),” no. June, pp. 1–43, 2013.

[12] vhost-user, https://github.com/qemu/qemu/blob/master/docs/specs/vhost-user.txt

[13] Open vSwitch, http://openvswitch.org