MIKELANGELO

D4.1
The First Report on I/O Aspects

Workpackage	4	Guest Operating System	
Author(s)	Shiqing Fan		Huawei
	Holm Rauchfuss		Huawei
	Nadav Har'El		ScyllaDB
Reviewer	Bastian Koller		HLRS
Reviewer	Philipp Wieder		GWDG
Dissemination Level	Public		

Date	Author	Comments	Version	Status
2015-11-03	Shiqing Fan	Initial document structure	V0.0	Draft
2015-12-01	Shiqing Fan	Initial draft version	V0.1	Draft
2015-12-11	Holm Rauchfuss	Further modifications	V0.2	Draft
2015-12-11	Shiqing Fan	Ready for review	V1.0	Review
2015-12-22	Shiqing Fan	Ready for submission	V2.0	Final

Executive Summary

In this deliverable, we present a first report on the I/O aspects, which have been designed and implemented as part of task T4.1 in the first year of MIKELANGELO project. We focus mainly on the lightweight RDMA virtualization solution called vRDMA and introduce briefly the new system programing API called Seastar, which is discussed in more details in report D4.4 ("OSv - Guest Operating System - First Version"). This work is going to develop techniques and mechanisms accelerating the virtual I/O and improving the scalability of multiple virtual machines running on a multi-core host. As a consequence this reduces the performance overhead incurred by virtualization and makes Cloud and High Performance Computing significantly more efficient.

The vRDMA solution aims to disrupt the overhead barrier preventing HPC Cloud adoption, i.e. HPC applications may have close to bare metal performance, but with all of the benefits of the Cloud: cost-efficiency and flexibility. On the other hand, use cases in the areas of Big Data and Cloud benefit from this virtualized infrastructure improvement as it not only provides lower overhead for normal socket-based interfaces, but it also enables programming models relying on RDMA (i.e. InfiniBand verbs API). Applications may run in a virtualized environment, e.g. an HPC Cloud, for flexibility, agility and costs, but meanwhile still retain high bandwidth and low latency for I/O.

All this enables Small Medium Enterprises (SMEs) and other companies that are currently unable to deploy their own HPC infrastructure to explore and scale their workloads to cloud providers, thus accelerating time to market and improving competitiveness.

The work plan for the I/O virtualization on RDMA protocol is divided into three basic steps, and each step includes the design and implementation of one prototype for different use cases and scenarios. The first project year is mainly focusing on the initial startup of the technical research, architecture design of the prototypes and the implementation of the first prototype. The complexity and difficulty of implementing prototype II and III are much higher than prototype I, so they are planned for the project year two and three respectively, and they will benefit from the valuable knowledge and experience of implementing prototype I.

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

Table of Tables

1 Introduction

The contents of this deliverable have been structured in several sections. Section 1 briefly introduces the basic design ideas of vRDMA and Seestar. In Section 2, we focus on the vRDMA design prototypes, which have been already discussed in M8 deliverable D2.13 ("The first sKVM hypervisor architecture") and M9 Deliverable D2.16 ("The First OSv Guest Operating System MIKELANGELO Architecture"). The actual implementation of prototype I is discussed deeply in Section 3, including the open source interfaces and packages and how they are integrated and configured. We present initial performance test results in Section 4 and compare several test cases with prototype I, which shows improvements of prototype I comparing to the other test cases. Section 5 concludes with a summary.

1.1 A lightweight paravirtualized RDMA solution

Remote Direct Memory Access (RDMA) interconnects are now widely being considered in deployments of large distributed data processing systems due to the nature of its high bandwidth, low latency and kernel bypass. In order to benefit from such interconnects in a virtualization environment, the RDMA I/O needs to be virtualized with a technology that will still have the RDMA advantages, but will minimize the virtualization disadvantages and impact. Single Root I/O Virtualization (SR-IOV) gives the possibility to share the RDMA device between multiple Virtual Machines (VMs) on the same host and to achieve performance close to bare-metal, but unfortunately it requires device and platform support, which introduces higher cost for development and maintenance both in guest and host. In order to have near bare-metal performance and still have low development and maintenance cost, solutions of paravirtualization have been proposed, where the control and data paths are separated to allow virtualizing RDMA capable Network Interface Controllers (NICs) entirely in software while still get near bare-metal performance.

A typical design or implementation of I/O paravirtualization consists of two major components, frontend driver and backend driver, and additionally a Virtual Switch may also be necessary for the infrastructure of multiple hosts.

A frontend driver emulates a virtual device for the guest OS, hiding the complexity of the virtualized infrastructure from the application (and guest OS) and offloading design complexity to the host side with the backend driver. A guest application then uses conventional communication API, such as socket for TCP/IP or InfiniBand (IB) verbs [2], for RDMA. The communication calls are directly processed in the frontend driver, then eventually translated and forwarded to the backend driver in the host.

For both HPC and Clouds with network interconnects supporting InfiniBand or Ethernet (to be more precise, RDMA over Converged Ethernet (RoCE)), we provide a lightweight RDMA virtualization solution, based on a virtio based virtual device and driver in the guest. It provides different network interfaces for the guest application and drives the communication over a RoCE network or uses shared memory within the same host. The main goal of the virtio-rdma is to improve the communication speed by RDMA and shared memory protocols, and to support guest applications using InfiniBand verbs or socket interface for inter-VM communications on the same host or between different hosts.

1.2 Seastar

There is much we can do to improve the performance of existing I/O-intensive applications, by improving the hypervisor (see D3.1, "The first Super KVM - Fast virtual I/O hypervisor"), the VM's kernel (see D4.4, "OSv - Guest Operating System – first version"), or both (as we present with vRDMA described in this document in Section 2 and 3). However, when we looked into the performance a certain kind of applications commonly used on the cloud - *asynchronous server applications* such as Memcached, Httpd and Cassandra, we discovered something interesting: What is most holding back the performance and scalability (towards many cores) of these applications is their use of aging operating-system APIs and program design principles. The implementation of these operating-system APIs can be improved to some extent (and we did this in MIKELANGELO with OSv, see D4.4), but even more radical improvements to these applications' performance can be achieved by completely redesigning the applications and the APIs they use. Seastar is such a newly-designed library for writing asynchronous I/O-intensive (network and disk) server applications aimed to replace the traditional operating-system APIs (sockets, threads, locks, etc.) and promote a style of server programming (*share-nothing* and *future-based asynchronous programming*) which is much more efficient on today's many-core systems than traditional server programming techniques.

In this report, we will mainly focus on the paravirtualized RDMA solution and the implementation of its first prototype. The details for Seastar, including the preliminary tutorial for writing modern Cloud applications could be found in M12 deliverable D4.4 ("OSv - Guest Operating System - First Version").

2 Design Prototypes

The implementation of the lightweight RDMA virtualization is composed of three prototypes as discussed in M8 deliverable D2.13 ("The first sKVM hypervisor architecture") and M9 deliverable D2.16 ("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.

Figure 1 shows an abstraction of these prototypes. The virtio-rdma is a frontend driver on the guest, which takes care of the communication requests from the guest application to the underlying hardware. It drives ivshmem [3], a user-level data movement library, to provide a shared memory communication mechanism.

A virtio-net [8] virtual device is used for communication between the guest and the virtual switch (prototype I and III). It may also support the InfiniBand verbs interface [2] by paravirtualizing the InfiniBand driver on the guest (prototype II), and the verb calls will be processed on the host with the help of DPDK rNIC PMD (Data Plane Development Kit RDMA Poll Mode Driver) [4]. A virtual switch based on the Open VSwitch implementation is used for managing and forwarding packets between the host and the guest.

The InfiniBand driver is supported and used based on the request from the guest and passed by DPDK rNIC PMD. RDMA memory regions, including Completion Queues (CQs), Work Requests (WRs) and Queue Pairs (QPs), are mapped on the host and shared between the guest application and the RDMA device, in order to increase the communication speed, i.e. kernel bypass. WRs are the communication requests with information of the peers and data. QPs are the actual data that are going to be transmitted. CQs are queues of events for notifying data arriving or completion of the communication.

An RDMA Controller module allows RDMA memory regions to be processed by the guest, either by guest applications (Prototype II) or by the virtio-rdma frontend driver (Prototype III). It then forwards the InfiniBand verb requests from the guest directly to the RDMA rNIC driver.

The vhost-user [6] 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.

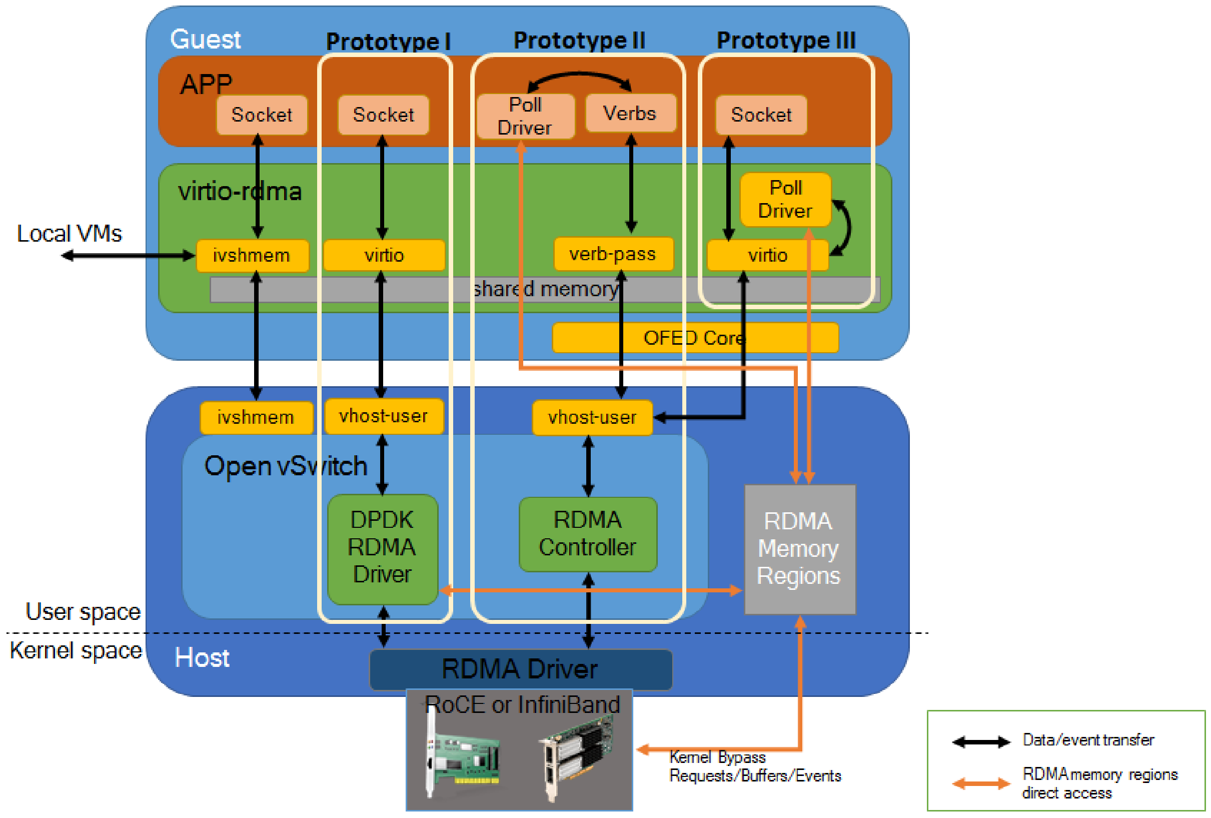

Figure 1: Architecture overview of the design prototypes

The main difference of the three prototypes lies in the interface that they support to the upper layer and in the position of the poll mode driver (PMD). Design prototype I supports socket API and uses the DPDK RDMA PMD directly on the host, which is integrated with Open vSwitch. It has the easiest implementation as most of the modules in this case are offered as open source projects that can be directly used and integrated. The communication buffers are mapped between the frontend driver and the RDMA device, and WRs and CQs are processed in the backend driver. The network API calls are translated into RDMA verbs for the device by the DPDK RDMA PMD. Processing these jobs in the backend driver will introduce a lot of overhead, as every single operation requires switching between the guest and the host many times, e.g. forwarding control commands and performing buffer operations. Additionally, processing completion events on the host is also considered as a slow path, due to the fact that events have to be passed through an event queue or ring buffer via the vhost-user. This removes completely the benefits of polling on the completion events. However, prototype I is important, not only because it provides a fast and easy solution of RDMA virtualization supporting socket applications in the guest, but also because that the implementation of prototype II and III will directly benefit from the knowledge and experience learnt from prototype I.

Design prototype II supports the RDMA verbs API, and the application implements and runs the PMD in the guest. Theoretically, it has the best performance, because communication buffers and completion events are managed and processed directly by the guest application. The involvement of the host is only for passing verbs to the RDMA device.

The main difference between prototype I and III is the position of the Poll Mode Driver. Both of them are aiming at supporting applications with socket API. But we may still have a choice between them based on the application features and requirements in order to have the best performance. For example, prototype I only runs one PMD for all the virtual machines on the same host, which means the overhead of consumption of hardware resources is minimum. As a result, it will have better performance when the guest application does not have heavy communication requests, where only one PMD is sufficient. On the other hand, prototype III is more efficient and optimal for heavy communication workload of the guest application, because the frontend driver on each guest OS runs its own PMD, which runs on a full virtual CPU. By polling on each virtual machine without interfering the host (data communication through RDMA memory regions), the overall performance is dramatically improved.

In the first year of the MIKELANGELO project, prototype I has been taken as the starting point and is implemented according to the project plan. Prototype II is planned for project year two, and prototype III is planned for year three. The final output of the RDMA virtualization will be the combination of prototype II and III, which will support both socket and verbs API simultaneously. The guest application will be able to run without any modification and without knowing the underlying virtual driver functionalities. More details on the implementation, performance tests and evaluation of prototype I will be discussed in Section 3 and 4.

3 Implementation

In this section, we focus on the implementation of prototype I. All interfaces that are used and the ways they are integrated are described.

3.1 Interfaces on the Guest

virtio-net

virtio-net [8] is an I/O virtualization PCI (Peripheral Component Interconnect) network device for guest OS. It was developed as a Linux KVM paravirtualized method for communicating network packets between host and guest. It is also available in OSv.

It provides multiqueue support as an approach to scale the network performance with the number of vCPUs by allowing them to transfer packets through more than one virtqueue pair at a time.

In singlequeue 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.

Virtio-net is used together with vhost-user in prototype I for the communication between guest and host as described above, with multiqueue support for vhost-user being enabled in upcoming Open vSwitch release [9]. The limitation of using singlequeue also exists in our prototype I implementation. The performance results presented in Section 4 should be significantly improved with the multiqueue support.

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 QEMU's 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 guest is a virtual device that accepts the up-layer sockets requests and communicate to the ivshmem server on the host.

3.2 Implementation and Integration

The basic implementation and integration instructions are attached in Appendix of this document. These instructions have been tested and used to build up a local testbed, and initial performance tests were run, which are presented in Section 4.

A more advanced testbed has been built up on a HPC cluster at MIKELANGELO partner HLRS. The implementation is the same as the local testbed, but special configurations were required due to the usage of NFS. The details of HPC integration has been described in M12 deliverable D5.4 (“First report on the Integration of sKVM and OSv with HPC”).

4 Accomplishments

The implementation and integration on a local testbed has been accomplished. This section will mainly focus on the test methodology and the performance results.

4.1 Test Environment

The initial performance tests were done on a testbed system consisting of two servers, each running 2 virtual machines. These two virtual machines are able to communicate through Ethernet or RoCE network interconnects between the hosts or through vhost-user inside the host.

The server is an HP ProDesk 600 with a 4-core Intel i7-4790 processor, and 16GB RAM memory with the host OS being Ubuntu 14.04 LTS server edition.

The host hypervisor is KVM/QEMU, and the VMs run guest OS, each configured with two vCPU (assigned to two isolated/dedicated cores on the host) and 2GB of memory.

An additional HPC testbed, provided by HLRS and described in D2.19 ("The first MIKELANGELO architecture"), will be used to further evaluate the performance gains.

The test benchmark is NetPIPE [5], a protocol independent network performance evaluator. It performs ping-pong test between two processes either over network or SMP with increasing message sizes. This benchmark has full support of MPI-2 application [7], thus it is commonly used to evaluate performance of an HPC environment. NetPIPE was run based on Open MPI 1.10.0, so we were able to use different protocols on the host easily, as shown in Figure 2.

There are two different network adapters that are configured on both hosts and used for the tests. The ConnectX-3 VPI adapter with FDR Dual-Port has up to 56Gb/s InfiniBand or 40 Gigabit Ethernet for RoCE per port. It uses PCI Express 3.0 (up to 8GT/s) and paravirtualized with the help of the design prototype I. In our tests, only RoCE mode of the InfiniBand adapter is used .The other one on the host is a 1Gb I217-LM Ethernet adapter.

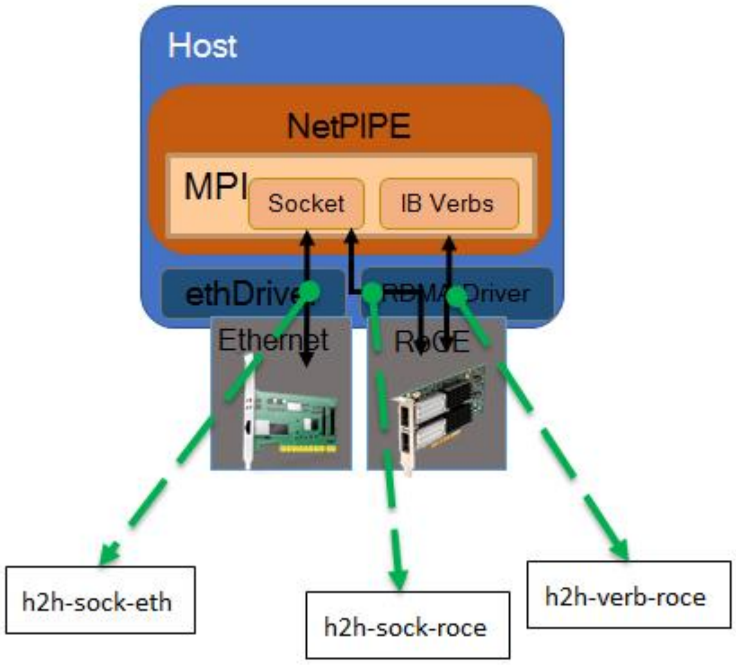

Figure 2: Test cases on the host

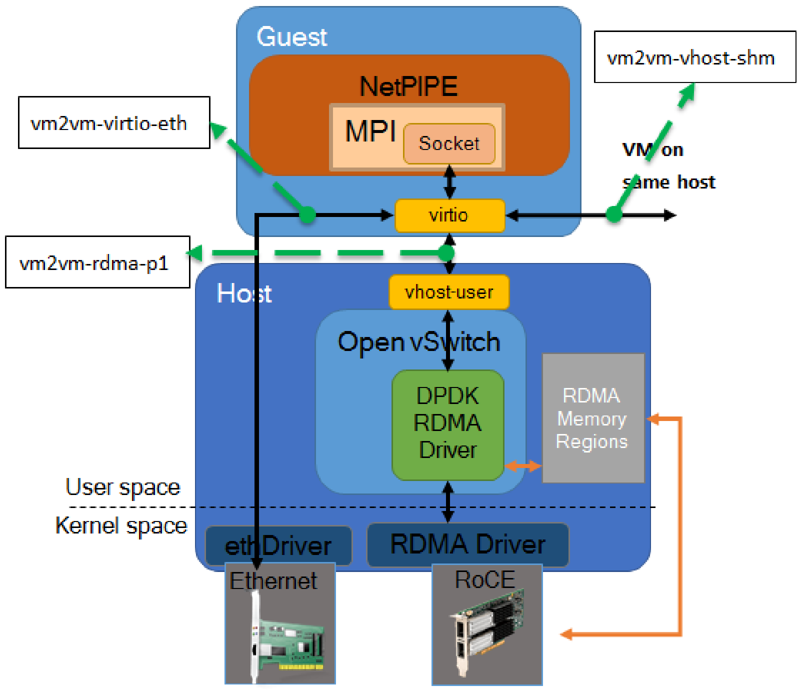

Figure 3: Test cases on the guest

Table 1 explains the test cases that are presented in Figure 2 and Figure 3:

Test name	**Description**
vm2vm-vhost-shm	VM to VM communication on the same host using shared memory based on DPDK vhost-user implementation.
vm2vm-virtio-eth	VM to VM communication on two hosts via Ethernet interconnect with virtio-net bridging mode on the local ethernet port (not utilizing vhost-user).
vm2vm-rdma-p1	VM to VM on two hosts via RoCE interconnect, using vRDMA prototype I.[1]
h2h-verb-roce	Host to host with InfiniBand verbs API over RoCE interconnect.
h2h-sock-roce	Host to host with socket API over RoCE interconnect.
h2h-sock-eth	Host to host with socket API over TCP/IP and Ethernet

Table 1: Description of the test cases

The following list shows the software configurations on host and guest:

- Host:
  - Ubuntu 14.04
  - DPDK 2.1.0
  - Open vSwitch 2.4.0
  - QEMU 2.3.0
  - libvirt 1.2.19
  - Open MPI 1.10.0
  - NetPIPE 3.7.2
- Guest:
  - Ubuntu 14.04
  - Open MPI 1.10.0
  - NetPIPE 3.7.2

4.2 Initial Performance Results

The test results are presented and analyzed in three categories, i.e. overall bandwidth (Figure 4), overall run time (Figure 5) and latency (Figure 6).

[1] for vm2vm test cases, all guest applications use socket API

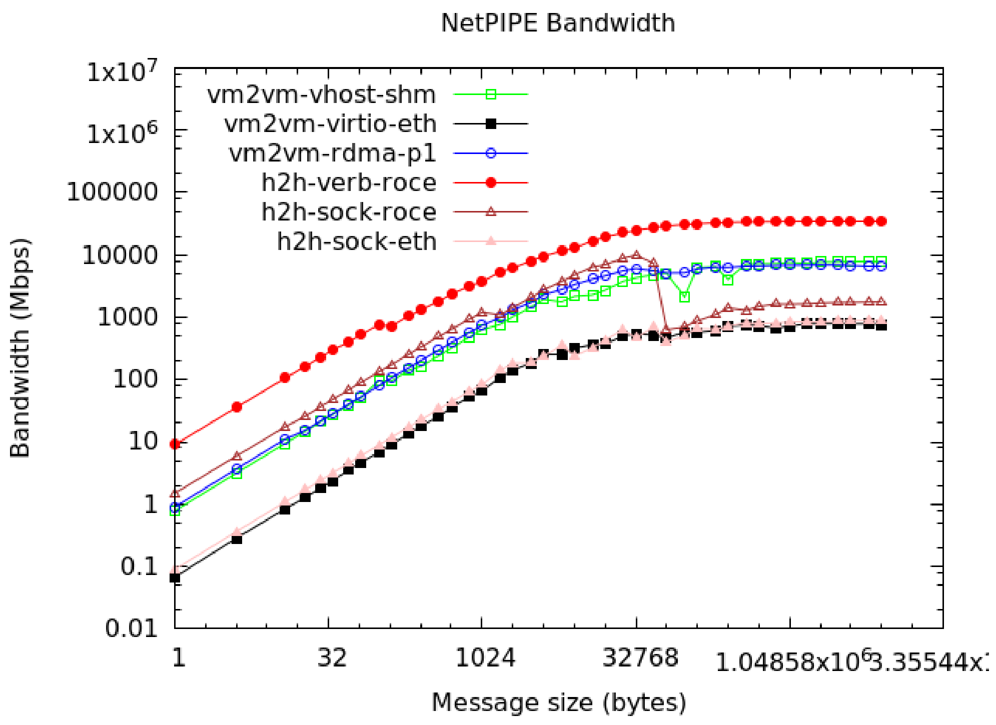

Figure 4: Initial Performance Results – Overall Bandwidth

In general, the test case using verbs API over RoCE interconnect has the best performance. The two test cases over Ethernet interconnect (vm2vm-virtio-eth and h2h-sock-eth) have a similar low performance compared to the other test cases. The result of the vRDMA prototype I (vm2vm-rdma-p1) shows very close performance to the cases of shared memory over vhost-user between VMs (vm2vm-vhost-shm) and the case of socket API over RoCE between hosts (h2h-sock-roce). vRDMA prototype I has achieved 20-25% of the bare-metal performance, i.e. test case using verbs API over RoCE (h2h-verb-roce). Theoretically, design prototype II of vRDAM will be the most efficient virtual I/O solution and get close to bare metal performance.

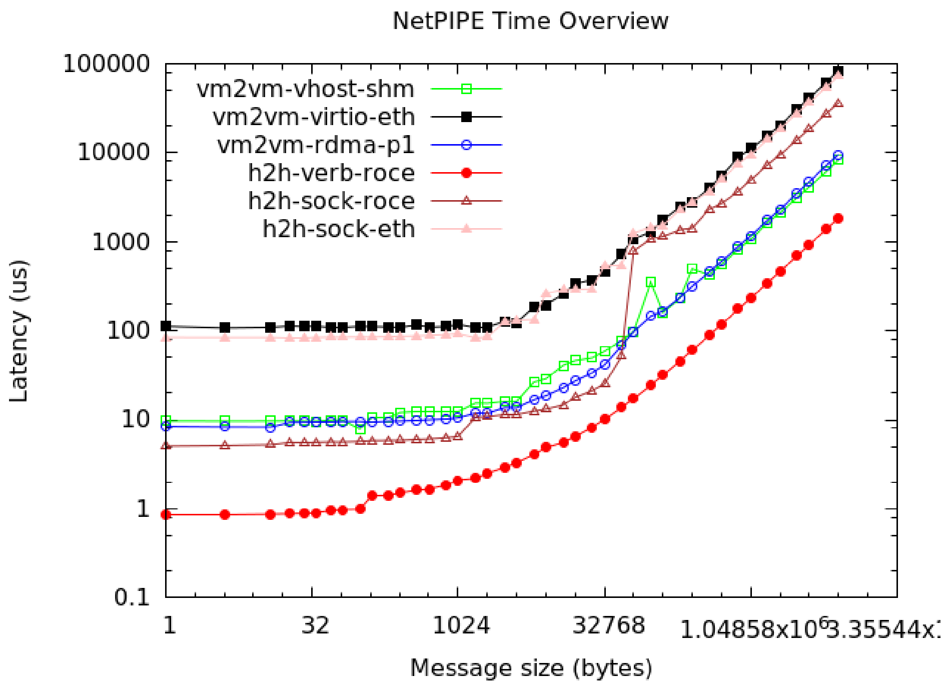

Figure 5: Initial Performance Results - Overview of the latencies at different message sizes

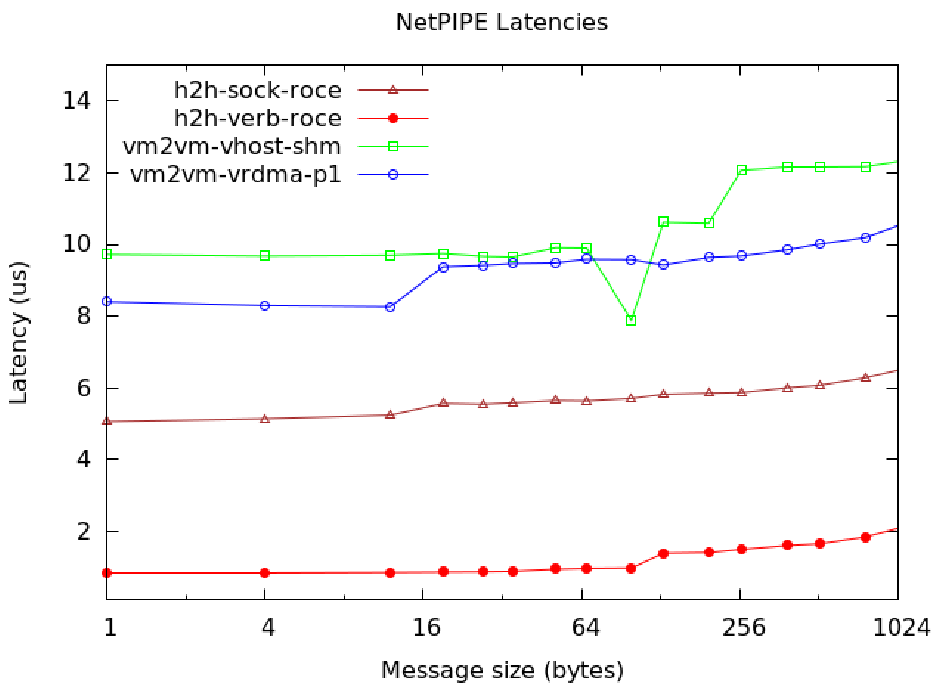

Figure 6: Initial Performance Results - Time for small messages (Latencies)

In the latency comparison in Figure 6, the verbs over RoCE case (h2h-verb-roce) has around 1.4 to 2.2 microseconds, while the vRDMA prototype I (vm2vm-vrdma-p1) has approximately 8.2 to 10.3 microseconds, which is about 1.6 to 2 microseconds faster than the case of shared memory via vhost-user (vm2vm-vhost-shm). The cases using Ethernet interface are not shown here, as they reach around 100 microseconds for the tests.

An initial evaluation on the test results has been made, and the reason why the vRDMA prototype I and shared memory via vhost-user have similar performance curve can be explained. When using the DPDK rNIC PMD, every single communication packet will be processed by the PMD on the host, no matter if it is a VM to VM communication on the same host or a host to host communication over the physical network. The key for improving the overall performance of vRDMA prototype I and the shared memory communication via vhost-user, is to optimize the PMD. This work will be addressed in the implementation of prototype II and III, and the further details and results will be presented in later deliverables in project year two and three. Furthermore, it is expected that the throughput performance for vRDMA prototype I increases as limits of the local testbed are eliminated on the more realistic HPC testbed (number and assignment of cores, memory size).

5 Concluding Remarks

In this report, we described the design prototypes of the lightweight RDMA paravirtualization solution. The implementation details and integration instructions for prototype I have been demonstrated, and an initial test has been presented. The initial test results showed promising performance comparing to traditional Ethernet, and the vRDMA prototype I implementation achieved about 25% of the bare-metal performance as a first step.

The evaluation of the bottleneck of vRDMA prototype I is still continuing and will be a one of the first tasks for the second project year. More optimization solutions will be analyzed and tested in the near future. Further synergy between the hypervisor-side improvements by IOcm and Seastar are also investigated.

In project year two, we will mainly focus on the implementation of prototype II, which uses a much more efficient way of paravirtualization to reduce the communication between the guest and host and improve the overall performance significantly. An updated I/O report on the latest achievements and results will be provided by the end of year two.

6 References and Applicable Documents

[1] The MIKELANGELO project, http://www.mikelangelo-project.eu/
[2] "InfiniBand Architecture Specification Volume 1," vol. 1, 2015.
[3] 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.
[4] I. Corporation, "Intel Data Plane Development Kit (Intel DPDK)," no. June, pp. 1–43, 2013.
[5] Q. O. Snell, A. R. Mikler, and J. L. Gustafson, "Netpipe: A network protocol independent performance evaluator," in In Proceedings of the IASTED International Conference on Intelligent Information Management and Systems, 1996.
[6] vhost-user, https://github.com/qemu/qemu/blob/master/docs/specs/vhost-user.txt
[7] MPI standards, http://www.mpi-forum.org/docs/docs.html
[8] R. Russell, "virtio: Towards a De-Facto Standard For Virtual I / O Devices," ACM SIGOPS Oper. Syst. Rev., vol. 42, pp. 95–103, 2008.
[9] Open vSwitch patch, http://openvswitch.org/pipermail/dev/2015-October/061413.html

7 Appendix - Instruction Manual

7.1 System Requirements

7.1.1 Hardware

The minimum hardware requirements for the basic setup on a single workstation are:

- Virtualization technology must be enabled in BIOS, e.g. VT-x for Intel CPUs and AMD-v for AMD CPUS
- Infiniband ConnectX-3 VPI or newer adapters that have RoCE support is required, and they are interconnected with proper cables and switches.

7.1.2 Software

Following is a list of software packages that have to be installed and configured in order to use RDMA virtualization prototype I:

- QEMU, at least 2.2, is required for huge page and vhost-user support.
- Open vSwitch, at least 2.4.0, is required for DPDK netdev support.
- DPDK, at least 2.1.0, is required for InfiniBand Poll Mode Driver support.
- libvirt, at least 1.2.19, is required for managing the VMs with easier configuration of vhost-user device.

7.2 Installation and Configuration Instructions

7.2.1 Setup Huge Pages and CPU Isolation

First of all, to gain the best performance, hyper-threading and power management have to be disabled in BIOS. Then CPU cores have to be selected and isolated on the same NUMA node, in order to avoid remote cache access. Command `lscpu` may be used to check the NUMA cores on the machine, and then isolate the CPU cores on the same NUMA node by appending the following parameters in the GRUB configuration file in `/etc/default/grub` at the line starting with "`GRUB_CMDLINE_LINUX=`":

```
iommu=pt intel_iommu=on default_hugepagesz=1G hugepagesz=1G hugepages=8 isolcpus=6-9
```

Update GRUB:

```
$ sudo update-grub
```

Reboot, then mount the huge pages:

```
$ sudo mkdir -p /dev/hugepages
$ sudo mount -t hugetlbfs -o pagesize=1G none /dev/hugepages
```

Test if the cores are isolated:

```
$ sudo apt-get install stress
$ stress -c 12 # num of cores on the system
```

Run "top" command in another terminal and press the "1" key, and it will show cores specified with `isolcpus` flag in GRUB (6-9 in this case) are at 0% of usage.

7.2.2 Install and Configure DPDK

Required version is 2.1.0
Download from this link: http://dpdk.org/browse/dpdk/snapshot/dpdk-2.1.0.tar.gz

Modify config/common_linuxapp, enable mlx4 and single lib support (vhost-user is enabled by default):

Change:	CONFIG_RTE_LIBRTE_MLX4_PMD=n
To:	CONFIG_RTE_LIBRTE_MLX4_PMD=y
Change:	CONFIG_RTE_BUILD_COMBINE_LIBS=n
To:	CONFIG_RTE_BUILD_COMBINE_LIBS=y

Run config and build command:

```
make install T=x86_64-ivshmem-linuxapp-gcc
```

This will generate the build directory (x86_64-ivshmem-linuxapp-gcc) in DPDK root directory.

Export several environment variables:

```
export RTE_SDK=/path/to/dpdk_root_dir
export RTE_TARGET=x86_64-ivshmem-linuxapp-gcc
```

7.2.3 Install and Configure Open vSwitch

Required version is 2.4.0.

Download from this link: http://openvswitch.org/releases/openvswitch-2.4.0.tar.gz
Open VSwitch doesn't need to be installed in the default location where requires root to access, but in order to cope with hugepages, root has to be used. Don't know how to avoid using root through all this stuff at moment.

If we want to use newer DPDK (>=2.1.0), then Open VSwitch has to link with "-ldpdk" not "-lintel_dpdk" anymore, because DPDK changed their library name.

Modify acinclude.m4 in Open Vswitch root directory:

Change:	DPDK_LIB="-lintel_dpdk"
To:	DPDK_LIB="-ldpdk"

Config and build (prefix may be changed, default is /usr/local):

```
$ ./boot.sh
$ ./configure --prefix=/usr/local --with-dpdk=$RTE_SDK/$RTE_TARGET \
            CFLAGS="-I $RTE_SDK/$RTE_TARGET/include -libverbs"
$ make
$ sudo make install
```

Export several environment variables if necessary (optional):

```
$ export OVS_DIR=/usr/local
$ export DB_SOCK=$OVS_DIR/var/run/openvswitch/db.sock
```

7.2.4 Install and Configure QEMU

Download the latest QEMU, 2.3.0 from this link:
http://wiki.qemu-project.org/download/qemu-2.3.0.tar.bz2

Solve dependencies:

```
$ sudo apt-get install libperl-dev libgtk2.0-dev zlib1g-dev
```

Install QEMU:

```
$ ./configure
$ make
$ sudo make install
```

7.2.5 Install and Configure libvirt

Required version is 1.2.19

Download from this link: http://libvirt.org/sources/libvirt-1.2.19.tar.gz

Solve dependencies:

```
$ sudo apt-get install libyajl-dev libdevmapper-dev \
               libpciaccess-dev libnl-3-dev libnl-route-3-dev \
               libxml2-dev dnsmasq
```

Config and install the package (prefix may be changed):

```
$ ./configure --prefix=/usr/local
$ make
$ sudo make install
```

After installation, the shared libraries need to be refreshed, otherwise the system will not able to find correct libvirt.so.0:

```
$ sudo ldconfig -v | grep libvirt
```

Please Note: Make sure the current user has enough permission to run libvirt daemon and virsh commands. Here we use root to ease the process, otherwise we cannot initialize a virtual bridge. And starting from this point, all following commands in the rest of this document are run in root context.

Add the user into the libvirtd user group, and start the daemon:

```
# usermod -G libvirtd -a root
# libvirtd -d
```

Configure the default network for libvirt by copying the default.xml from the libvirt source directory:

```
# cp /path/to/libvirt/src/network/default.xml \
   /usr/local/var/run/libvirt/network
# virsh net-define /usr/local/var/run/libvirt/network/default.xml
# virsh net-autostart default
# virsh net-start default
```

Import domains/VMs or install a new one from scratch based on the requirement.

7.3 Execution Instructions on Single Workstation

In this section, we focus on step-by-step instructions to set up a run-time configuration and to start the virtual environment for vRDMA on a single workstation. The run-time configuration for HPC is in section 7.4.

7.3.1 Start Open vSwitch

Load necessary kernel modules:

```
# insmod $RTE_SDK/lib/librte_vhost/eventfd_link/eventfd_link.ko
```

Load the kernel modules for DPDK:

```
# modprobe -a ib_uverbs mlx4_en mlx4_core mlx4_ib
```

Configure InfiniBand port into RoCE mode using:

```
# connectx_port_config
```

config the RoCE port name to dpdk* in file /etc/udev/rules.d/70-persistent-net.rules:

```
# PCI device 0x15b3:0x1003 (mlx4_core)
SUBSYSTEM=="net", ACTION=="add", DRIVERS=="?*", ATTR{address}=="f4:52:14:6f:5f:71",
ATTR{dev_id}=="0x0", ATTR{type}=="1", KERNEL=="eth*", NAME="dpdk0"
```

Note: Reboot may be required to refresh the interface name. To use dpdk port with Open VSwitch, the port name has to be dpdk*, like dpdk0, dpdk1 and so on.

Find out the InfiniBand port information (please also refer to dpdk doc in dpdk-2.1.0/doc/guides/nics/mlx4.rst):

```
# { \
    for intf in dpdk0 dpdk1 dpdk2 dpdk3; \
    do \
        (cd "/sys/class/net/${intf}/device/" && pwd -P); \
    done; \
} | \
sed -n 's,.*/\(.*\),-w \1,p'
```

This will return the bus information like:

```
-w                                                                    0000:02:00.0
-w                                                                    0000:02:00.0
-w 0000:06:00.0
-w 0000:06:00.0
```

These will be used as the parameter for starting Open VSwitch

Start libvirt daemon if it's not automatically done:

```
# libvirtd -d
```

Clean up and create the database:

```
mkdir -p $OVS_DIR/etc/openvswitch
mkdir -p $OVS_DIR/var/run/openvswitch
rm $OVS_DIR/etc/openvswitch/conf.db
ovsdb-tool create /usr/local/etc/openvswitch/conf.db \
       $OVS_DIR/share/openvswitch/vswitch.ovsschema
```

Start the Open VSwitch server without SSL:

```
# ovsdb-server --remote=punix:$DB_SOCK \
               --remote=db:Open_vSwitch,Open_vSwitch,manager_options \
               --pidfile --detach
# ovs-vsctl --db=unix:$DB_SOCK --no-wait init
# rm /usr/local/var/log/openvswitch/ovs-vswitchd.log
# rm /usr/local/var/run/openvswitch/dpdkvhost*


# rmmod openvswitch
# ovs-vswitchd --dpdk -c 0x1 -n 4 -w 0000:02:00.0 \
               --socket-mem 2048,0 -- unix:$DB_SOCK -pidfile \
               --detach --log-file
```

Create a virtual switch, dpdk port and vhost-user channel (one virtual switch on one host is enough, but if we want to run multiple VMs on a single host, then we need to create more dpdk ports and vhost-user channels for each VM):.

```
# ovs-vsctl add-br br0 -- set bridge br0 datapath_type=netdev
# ovs-vsctl add-port br0 dpdk0 -- set Interface dpdk0 type=dpdk
# ovs-vsctl add-port br0 dpdkvhost0 \
               -- set Interface dpdkvhost0 type=dpdkvhostuser
```

Tune the performance by set core mask for the pmd thread, for example on core 2 and 4 on the same NUMA node (in binary: 00010100):

```
ovs-vsctl set Open_vSwitch . other_config:pmd-cpu-mask=000014
```

7.3.2 Start the Virtual Machine

To add a vhost-user device, configure the xml file of the virtual machine (virsh edit vm-name), add following section in <devices>…</devices> section:

```
   <interface type='vhostuser'>
      <mac address='52:54:00:3b:83:1a'/>
      <source type='unix' path='/usr/local/var/run/openvswitch/dpdkvhost0' mode='client'/>
      <model type='virtio'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x03' function='0x0'/>
    </interface>
```

To enable the huge pages support, configure the xml file of the virtual machine, add following section in <domain>…</domain> section:

```
        <memoryBacking>
          <hugepages>
             <page size='1024' unit='MiB'/>
          </hugepages>
        </memoryBacking>


        <cpu>
          <numa>
            <cell id='0' cpus='0-1' memory='3064' unit='MiB' memAccess='shared'/>
          </numa>
        </cpu>
```

7.4 Special Configuration for HPC Environment

All instructions, which have been described in the previous sections, have been tested on a single workstation. When deploying the same configuration in an HPC environment, the same processes are still valid, but there are a few steps have to be done in differently to fit for the HPC requirements.

For HPC, software packages are normally installed on an NFS to be used by all the compute nodes, i.e. a one-time installation to be used everywhere. The use of NFS introduces additional complexities for integrating the design prototype I into the system. For example, all the compute nodes may use the same software installation, which normally results in access conflicts of the same context files or metadata that are stored in the installation directory with the same default file names.

7.4.1 Special configuration for Open vSwitch

For each command in Open vSwitch, a database socket has to be connected. The default socket is under `/<path/to/openvswitch>/var/run/openvswitch/db.sock`, which is usually an NFS path. In such case, Open vSwitch commands from different nodes would have conflicts to access the same socket.

The solution in this case is to specify different sockets for different nodes using their node names as the identifiers. We use several environment variables to avoid such conflicts that are shown in Table 2.

Variable Name	Variable Content (full path omitted)	Description
`NODENAME`	./.	Name of the node.
`OVS_DATABASE`	`conf-$NODENAME.db`	Full path to the database of OVS server.

OVS_SERVER_LOG	ovs-server-$NODENAME.log	Full path to the log file for OVS server.
OVS_DAEMON_LOG	ovs-daemon-$NODENAME.log	Full path to the log file for OVS daemon.
OVS_SERVER_PID	ovs-server-$NODENAME.pid	Full path to the PID file for OVS server.
OVS_DAEMON_PID	ovs-daemon-$NODENAME.pid	Full path to the PID file for OVS daemon.
DB_SOCK	ovs-db-$NODENAME.sock	Full path to the socket for OVS applications to connect to the database.
DPDKVHOST_NAME	dpdkvhost-$NODENAME	Name of the socket for OVS to connect to vhost-user.
DPDKVHOST	dpdkvhost-$NODENAME	Full path to the vhost-user socket.

Table 2: List and descriptions of the environment variables used for Open vSwitch (OVS)

7.4.2 Special Configuration for libvirt

Similarly, for each command in libvirt, a socket has to be used to connect with QEMU. The default socket is under `</path/to/libvirt>/var/run/libvirt-sock`, which is usually an NFS path. In such case, libvirt commands from different nodes would have conflicts to access the same socket.

The solution is also to specify different sockets for different nodes using their node names as the identities. We use several environment variables to avoid such conflicts. Table 3 shows a list of these t variables.

Variable Name	Variable Content (full path omitted)	Description
LIBVIRT_PID	libvirtd-$NODENAME.pid	Full path to the PID file for libvirt daemon.
LIBVIRT_CONFIG	libvirtd-$NODENAME.conf	Full path to the config file for libvirt daemon, which contains a different directory for libvirt-sock. The config file has to be created before libvirtd starts.
LIBVIRT_URI	libvirt-$NODENAME/libvirt-sock	Full path to a different directory for libvirt-sock.

Table 3: List and descriptions of the environment variables used for libvirt commands

7.4.3 Command Line Aliases for MIKELANGELO

When running the same command line on different compute nodes, in order to use different contexts for each node, different connections between QEMU and libvirt have to be used, which results a rather long command line for libvirt to use the correct socket. For example *virsh list*, following parameters have to be provided:

```
# virsh -c \
qemu+unix:///system?socket=/opt/libvirt/libvirt-1.2.19/var/run/libvirt-cn1/libvirt-sock \
list
```

Here, libvirt-sock is the connection socket between libvirt and QEMU, and it has to be stored in a directory specifically for this node, i.e. in `/opt/libvirt/libvirt-1.2.19/var/run/libvirt-cn1`, where `cn1` is the node name.

As a few of the commands related to Open vSwitch and libvirt have to follow this scenario, environment variables are used. For example, to run the equivalent command above, we are able to simply run:

```
# virsh -c $LIBVIRT_URI list
```

In order to further hide the complexities, command aliases have been applied by adding a prefix "mike-" to these commands. So the following command is identical to the previous one:

```
# mike-virsh list
```

This prefix rule has been applied to a few Open vSwitch and libvirt related commands, including *virsh*, *ovs-vsctl*, and *virt-manager*.

7.4.4 An example script for setting up in HPC Environment

The following script can be used to set up the configuration on each node at startup time. For example, assuming all the software and hardware are configured as described in Sections 7.1 and 7.2, the following command can be simply used instead of going through the instructions in Section 7.3:

```
# source start-vrdma.sh node1 10.0.0.1/16
```

The first parameter for the script is the node name or id, and the second parameter is the assigned IP address for the dpdk port, which is the same as the virtual bridge address.

```
#!/bin/sh
#
# start up script: start-vrdma.sh
#
export PATH=/opt/openmpi/openmpi-1.10.0/bin:/opt/libvirt/libvirt-
```

```
1.2.19/bin:/opt/libvirt/libvirt-1.2.19/sbin:/opt/openVSwich/bin:/opt/qemu/qemu-
2.3.0/bin:$PATH
export LD_LIBRARY_PATH=/opt/openmpi/openmpi-1.10.0/lib:/opt/libvirt/libvirt-
1.2.19/lib:/opt/openVSwich/lib:/opt/qemu/qemu-2.3.0/lib:$LD_LIBRARY_PATH

export NODENAME=$1
export IP_ADDR=$2

export DPDK_DIR=/opt/dpdk/dpdk-2.1.0
export RTE_SDK=$DPDK_DIR
export RTE_TARGET=x86_64-ivshmem-linuxapp-gcc
export DPDK_BUILD=$DPDK_DIR/$RTE_TARGET
export OVS_DATABASE=/opt/openVSwich/etc/openvswitch/conf-$NODENAME.db
export OVS_SERVER_LOG=/opt/openVSwich/var/log/openvswitch/ovs-server-$NODENAME.log
export OVS_DAEMON_LOG=/opt/openVSwich/var/log/openvswitch/ovs-daemon-$NODENAME.log
export OVS_SERVER_PID=/opt/openVSwich/var/log/openvswitch/ovs-server-$NODENAME.pid
export OVS_DAEMON_PID=/opt/openVSwich/var/log/openvswitch/ovs-daemon-$NODENAME.pid
export DB_SOCK=/opt/openVSwich/var/run/openvswitch/ovs-db-$NODENAME.sock
export DPDKVHOST_NAME=dpdkvhost-$NODENAME
export DPDKVHOST=/opt/openVSwich/var/run/openvswitch/dpdkvhost-$NODENAME
export LIBVIRT_PID=/opt/libvirt/libvirt-1.2.19/var/run/libvirtd-$NODENAME.pid
export LIBVIRT_CONFIG=/opt/libvirt/libvirt-1.2.19/etc/libvirt/libvirtd-$NODENAME.conf
export LIBVIRT_URI=qemu+unix:///system?socket=/opt/libvirt/libvirt-1.2.19/var/run/libvirt-
$NODENAME/libvirt-sock

# hide a bit of the complexity using alias
alias mike-virsh="virsh -c $LIBVIRT_URI"
alias mike-ovs-vsctl="ovs-vsctl --db=unix:$DB_SOCK"
alias mike-virt-manager="virt-manager -c $LIBVIRT_URI"

mkdir -p /dev/hugepages
umount /dev/hugepages
echo 4096 > /sys/devices/system/node/node0/hugepages/hugepages-2048kB/nr_hugepages
echo 4096 > /sys/devices/system/node/node1/hugepages/hugepages-2048kB/nr_hugepages
mount -t hugetlbfs none /dev/hugepages

killall libvirtd
cp /opt/libvirt/libvirt-1.2.19/etc/libvirt/libvirtd.conf $LIBVIRT_CONFIG

sed -i -- "s/#unix_sock_dir = \"\/var\/run\/libvirt\"/unix_sock_dir =
\"\/opt\/libvirt\/libvirt-1.2.19\/var\/run\/libvirt-$NODENAME\"/g" $LIBVIRT_CONFIG

libvirtd -d -p $LIBVIRT_PID -f $LIBVIRT_CONFIG

# clean up and create the database
rm $OVS_DATABASE

ovsdb-tool create $OVS_DATABASE \
           /opt/openVSwich/share/openvswitch/vswitch.ovsschema

# as we use DPDK for the datapath, this kernel module has to be unloaded to avoid conflicts
rmmod openvswitch
rm $OVS_SERVER_LOG
rm $OVS_DAEMON_LOG
rm $OVS_SERVER_PID
rm $OVS_DAEMON_PID
rm $DB_SOCK
rm $DPDKVHOST

# no SSL
ovsdb-server $OVS_DATABASE --remote=punix:$DB_SOCK \
             --remote=db:Open_vSwitch,Open_vSwitch,manager_options \
             --pidfile=$OVS_SERVER_PID --detach \
             --log-file=$OVS_SERVER_LOG
mike-ovs-vsctl --no-wait init

# start ovs daemon for Mellanox
# multiple pci devices can be added via "-w 0000:0x:00.0"
ovs-vswitchd --dpdk -c 0x1 -n 4 -w 0000:05:00.0 --socket-mem 2048,0 \
             -- unix:$DB_SOCK --pidfile=$OVS_DAEMON_PID --detach \
             --log-file=$OVS_DAEMON_LOG
```

```
# add the virtual bridge
mike-ovs-vsctl --no-wait add-br br1 -- set bridge br1 datapath_type=netdev

# add first pair, a dpdk port and a vhost-user port, add default flows
mike-ovs-vsctl --no-wait add-port br1 dpdk1 -- set Interface dpdk1 type=dpdk
mike-ovs-vsctl --no-wait add-port br1 $DPDKVHOST_NAME \
               -- set Interface $DPDKVHOST_NAME type=dpdkvhostuser

# if cpus are not isolated, the last command won't work, use following command instead.
taskset -acp 4 `pidof ovs-vswitchd`

# configure the ip addresses of the bridge and ports
ifconfig br1 0
ifconfig br1 $IP_ADDR
#ifconfig dpdk0 0
ifconfig dpdk1 0
ifconfig dpdk1 promisc up
```

In order to clean up everything for a restart of the virtual bridge and libvirt daemon, a stop script can be used:

```
# source stop-vrdma.sh
```

The script has the following content:

```
#!/bin/sh

#
# stop script: stop-vrdma.sh
#

mike-ovs-vsctl --no-wait del-br br1

killall ovs-vswitchd
killall ovsdb-server

ifconfig dpdk1 $IP_ADDR
```