Towards a Lightweight RDMA Para-Virtualization for HPC

Shiqing Fan
Huawei Technologies
German Research Center
shiqing.fan@huawei.com

Fang Chen
Huawei Technologies
German Research Center
fang.chen1@huawei.com

Holm Rauchfuss
Huawei Technologies
German Research Center
holm.rauchfuss@huawei.com

Nadav Har'El
ScyllaDB
nyh@scylladb.com

Uwe Schilling
University of Stuttgart
HLRS Stuttgart
schilling@hlrs.de

Nico Struckmann
University of Stuttgart
HLRS Stuttgart
struckmann@hlrs.de

Keywords

VirtualizationvirtIO; RDMA; HPC; unikernel

ABSTRACT

Virtualization has gained increasing attention in the recent High Performance Computing (HPC) development. While HPC provides scalability and computing performance, HPC in the cloud benefits in addition from the agility and flexibility that virtualization brings. One of the major challenges of HPC in virtualized environments is RDMA virtualization. Existing implementations of RDMA virtualization focused on supporting VMs running Linux. However, HPC workloads rarely need a full-blown Linux OS. Compared to traditional Linux OS, emerging *Library OSes*, such as *OSv*, are becoming popular choices as they provide efficient, portable and lightweight cloud images. To enable virtualized RDMA for lightweight library OSes, drivers and interfaces must be re-designed to accommodate the underlying virtual devices. In this paper we present a novel design, the *virtio-rdma* driver for *OSv*, which aims to provide RDMA para-virtualization for lightweight library OS. We compare this new design with existing implementations for Linux, and analyze the advantages of *virtio-rdma*'s architecture, its ease of migration to different operating systems, and the potential for performance improvement. We also propose a solution for integrating this para-virtualized driver into HPC platforms, enabling HPC application users to deploy their use cases smoothly in a virtualized HPC environment.
ïż£

1. INTRODUCTION

Para-virtualization has been commonly used in virtualized environments to improve system efficiency and to optimize management workloads. In the era of High Performance Computing (HPC) and Big Data use cases, cloud providers and HPC centers focus more on developing para-virtualization solutions of fast and efficient I/O. Due to the nature of high bandwidth, low latency and kernel bypass, Remote Direct Memory Access (RDMA) [6] interconnects play an important role for the I/O efficiency, and it has been widely deployed in HPC and data centers as an I/O performance booster. To benefit from these RDMA advantages in virtualized HPC environment, network communication supporting InfiniBand and RDMA over Converged Ethernet (RoCE) [5] must be enabled for the underlying virtualized devices.

There are a few existing solutions for RDMA virtualization, e.g. *vRDMA* [1] from VMware and *HyV* [2],a hybrid I/O virtualization framework for RDMA-capable network interfaces, from IBM. However, none of them is applicable for virtualization on HPC. *vRDMA* is only available for *VMware ESXi* guest, and it is not open source based and not free. *HyV* supports only for Linux kernel 3.13, and it relies heavily on Linux kernel drivers, which tightly couples Linux host and guest, excluding the usage of lightweight library OSes [9].

To enable such communications for virtualized devices with *OSv*, a lightweight, fast and simple library OS for Cloud, we designed a new para-virtualized frontend driver, *virtio-rdma*, for RDMA-capable fabrics. This solution aims to disrupt the overhead barrier preventing HPC Cloud adoption, enable HPC applications to run in virtual machines with a performance comparable to bare metal, and bring all the benefits from the Cloud. The *virtio-rdma* frontend driver is designed to support also shared memory communication for the virtual machines (VM) on the same host. By switching the protocols automatically in *virtio-rdma*, user application uses only the standard RDMA API to accomplish both inter-host and intra-host communications.

On the other hand, our design also includes a solution to use *OSv* and *virtio-rdma* in HPC environment. By extending *Torque* [3], necessary environment settings are configured to launch *OSv* and its components. The job submitting procedure is the same as on a normal HPC platform, except the job command line needs simple adoptions.

This paper is structured as follows: section 2 introduces the fundamental work for this design; section 3 presents the details of the *virtio-rdma*, including its components, capabilities and advantages; section 4 shows the basic integration solution for running HPC jobs with *OSv* and *virtio-rdma*; section 5 concludes the paper and describes the current states of the implementation and future plan.

XXXXXX XXXXX
© 2017 ACM. ISBN 123-4567-24-567/08/06. . . $15.00
DOI: 10.475/123_4

2. BACKGROUNDS

In this section, we introduce several key frameworks and projects which serve as the foundation for our RDMA para-virtualization.

Traditional Root I/O Virtualization (SR-IOV) shares RDMA devices between multiple virtual machines, it also introduces high development and maintenance costs. SR-IOV is strictly dependent on the choice of hardware devices, making migration difficult when switching to a new Network Interface Controller (NIC). Unlike SR-IOV, para-virtualization RDMA solutions are more flexible, and it uses standard APIs therefore it can be applied with different NICs. Moreover, it allows VMs with direct access to RDMA memory regions and avoids moving communication data around between host and guest.

2.1 Remote Direct Memory Access

Remote Direct Memory Access (RDMA) is the ability to directly access another system's memory without involving the CPU on that remote system. Two major technologies support RDMA: InfiniBand [4] and RoCE. InfiniBand provides support for native RDMA, providing highly efficient and low latency interconnect technology. For Ethernet based network connections, RoCE provides true RDMA semantics, serving as an efficient Ethernet solution today.

The RDMA verbs API [7], is an abstract interface to an RDMA enabled NIC (RNIC). It is a de facto standard software stack defined by *OpenFabrics Enterprise Distribution (OFED)*. OFED stack aims to develop open-source software for RDMA and kernel bypass applications. The interface provides access to the RNIC queuing and memory management resources and is normally implemented as a combination of the RNIC by the device vendors. Therefore, the level of abstraction for our RDMA para-virtualization will be the verbs API to be vendor independent. From an architecture point of view, a virtualized RDMA driver consist of a backend driver at host side which communicates with the actual physical device, and a frontend driver at the guest level. The communication between frontend and backend drivers are then carried out using Hypercalls.

2.2 OSv and virtio

Our implementation uses the *OSv* [8] operating system on each VM. *OSv* was designed and optimized specifically for running a single application on a virtual machine in the cloud. We note that the modern cloud provides features, such as isolation, hardware abstraction, and management, which were traditionally provided by operating systems. By not duplicating these features, *OSv* is simpler, smaller and faster than traditional operating systems such as Linux, and helps reduce the overhead of virtualization. It is basically derived from FreeBSD and has been largely re-implemented under C++11 standard. Each VM has a single copy of *OSv*, and each VM runs a single application.

OSv uses a single address space for the kernel and a single application's threads, and has been considered a *library OS* [9] or more recently, a *unikernel* [10]. By using the same page tables for all threads and the kernel, context switches are largely reduced. But unlike some other unikernels which support only a limited range of applications or hypervisors, *OSv* can run on most common hypervisors (KVM, Xen and VirtualBox), and run unmodified Linux applications; It also fully supports multi-threading, and VMs with multiple cores. Moreover, paging and memory mapping are supported in *OSv* via *mmap* API.

As *OSv* runs only in VMs, it does not need to implement drivers for a huge variety of real hardware such as network cards. Rather, it needs to support just a few para-virtual devices provided by the hypervisor. The KVM hypervisor uses the *virtio* [11] protocol as a framework for all low-overhead guest I/O; The the para-virtual network driver (*virtio-net*) and disk driver (*virtio-blk*) are implemented using this protocol. With virtio, the performance-critical data path has minimal overheads such as guest-host context switches, while the host retains full control over the management of the device.

The present work adds a new driver to *OSv*, *virtio-rdma*. This uses the same virtio protocol to allow the guest to efficiently use the host's hardware RDMA capabilities with minimal overheads (such as guest-host context switches) while still giving the host full control with whom each guest can communicate.

2.3 Torque

Torque [3] is a portable batch system, which is used in many HPC platforms, e.g. clusters at High Performance Computing Center Stuttgart (HLRS). It queues, schedules and executes compute jobs inside clusters.

In order to execute the same jobs inside VMs, *Torque* has to be extended and modified to set up suitable environment for starting the guest OS and initializing the
drivers that to be loaded for running the jobs.

3. THE NEW PARA-VIRTUALIZATION DESIGN

In this section, we describe the new *virtio-rdma* design comparing with the existing designs, how it is capable to work for OSv and several advanced features. Our new design is based on HyV, adapting and extending it where needed. It makes extensive use of hypercalls, to facilitate direct communication between guest and host.

The general idea of the design architecture is shown in Figure 1. The guest application uses *socket* or *RDMA*. The *virtio-rdma* frontend driver will decide which communication protocol to be used based on the location of the peers. For example, the communication between VMs on the same host will use shared memory as a short cut for better performance. The communication between VMs on remote host will use the virtual *RDMA* path.

3.1 Frontend Driver

Theoretically, there are several different possible design schemes for RDMA para-virtualization frontend driver. It can be implemented in different layers of the guest OS, i.e. drivers in guest user space, providers in guest kernel space or both.

HyV [2] follows the rules of using the *OFED* user and kernel modules as is, and replacing a few kernel modules with its own *virtio* drivers in the guest kernel space. As shown in Figure 2, we present an example of querying device verb call showing the differences between the standard path and the *HyV* path. The *libibverbs* library exposes the verbs API to the application. The *libmlx4* library is the device provider in the user space, which is also known as a plugin to *libibverbs* library for Mellanox devices. For different RDMA devices, other plugin may be loaded. The

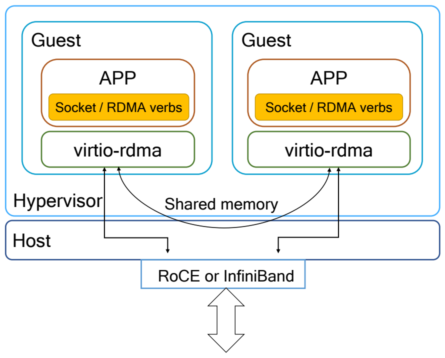


Figure 1: Architecture overview of *virtio-rdma*.

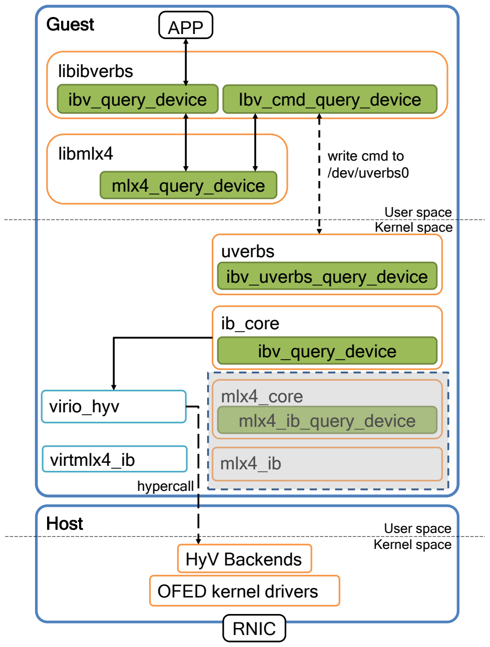


Figure 2: Workflow of calling *ibv_query_device* function using *HyV*.

libmlx4 library provides necessary hardware information to prepare the verb command in *libibverbs* library. The uverbs kernel module exposes a uverbs device to the user space, which handles the request command written by the registered clients. The uverbs kernel module transforms the command to the kernel context and passes it to the lower level kernel module, i.e. *ib_core*, *mlx4_core* and *mlx4_ib*, to perform the real hardware operation. *HyV* replaces *lx4_core* and *mlx4_ib* kernel modules with its own *virtio* drivers, i.e. *virtio_hyv* and *virtmlx4_ib*, to send hypercalls to the host instead of manipulating the real hardware. The corresponding host driver (*vhost_hyv*) handles the request and perform the actual hardware operation with the local *OFED* support.

The *HyV* guest drivers abstract the kernel verb calls that are able to work with *OFED* kernel modules, thus it highly relies on several unmodified *OFED* modules. When it comes to a library OS like OSv, implementing the same design become very difficult and even impossible due to the lack of *OFED* support. A library OS normally has only the minimum set of libraries that are required to support the targeting application, and there have no available support of *OFED* drivers for OSv or any other library OSes. Supporting the original *HyV* on *OSv* will involve extra work of porting the *OFED* modules and additional *FreeBSD* kernel implementations, which may end up to porting most of the *FreeBSD* kernel to *OSv*.

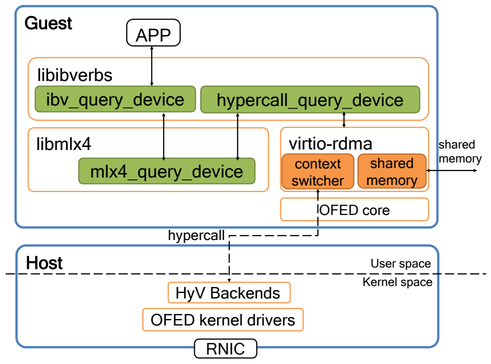


Figure 3: Workflow of calling *ibv_query_device* function using *virtio-rdma*.

However, implementing similar frontend drivers as *HyV* on *OSv* is still possible with even a much simpler architecture. For a library OS, the libraries are compiled and sealed with the application together into fixed images that can be run directly with a hypervisor. The created images are constructed with single address space, that the need for transmitting data between user space and kernel space is not necessary. This specific feature of allowing direct access to hardware resources without context switches does not only improve the performance, but also allows us to get rid of the dependencies of the *OFED* kernel modules.

The new *virtio-rdma* driver is designed to work with minimum support of *OFED* kernel providers or drivers, as shown in Figure 3. We keep two user space libraries: *libmlx4* provides the basic hardware information, e.g. vendor ID, device ID and hardware specific parameter format; *libibverbs* exposes the verbs API to the user application, and compose and send the hypercall message to the host driver. Both of them have been simplified: only the necessary fundamental support is kept; the available implementation in *OSv* has higher priority to be reused and to replace the *OFED* implementation.

At moment *virtio-rdma* supports only InfiniBand devices by dynamically loading *libmlx4* plugin. But this can be extended by supporting more device providers in the guest, and replace *libmlx4* at runtime.

3.2 Backend Driver

On the host side, we take the advantage of using *HyV vhost* driver, and the communication contains the same data structure as *HyV* uses. The work on the host side involves only porting the *HyV* host drivers to the targeting Linux version [1]. The new frontend driver provides the similar hypercall functionality as *HyV*, but with entirely new implementation based on the *OSv* implementation, which is mostly C++11 standard based and the *virtio* API is defined

[1] *HyV* was initially implemented for Linux Kernel version 3.13. Our project is targeting 3.18 for the host OS.

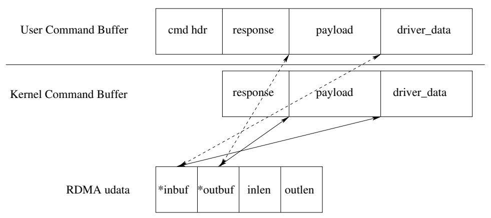


Figure 4: Context switch: command buffer and udata.

differently as on Linux and FreeBSD.

3.3 Context Switch

The most significant buffer that requires frequent user to kernel switch is the *RDMA* command buffer, which contains a command header, response buffer, payload buffer and driver data. Normally, when calling a verbs API from user space, a user command buffer is initialilzed in *libibverbs* and copied to the kernel provider, which creates a *udata* structure based on the response and payload buffers. The *udata* is then pushed to the *RDMA* kernel call. After the call is finished, the kernel provider copies *udata* back to user space. *HyV* doesn't change this process on the Linux guest.

In order to be compatible with *HyV* backend driver for using *virtio-rdma*, the hypercall parameters, especially the user command buffer, have to be converted to the kernel format. As OSv uses single address space implementation, we are able to avoid many context switches for handling the *RDMA* command buffer, by diretlly mapping *udata* into user command buffer, as shown in Figure 4.

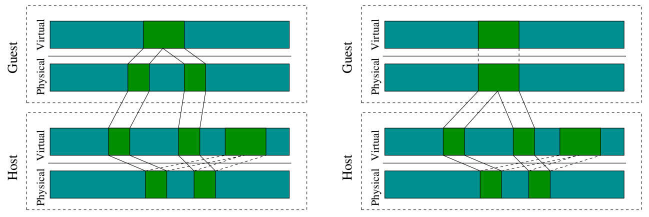
As we directly map the kernel objects to the user space in *OSv*, no kernel provider will be required, except a few *OFED* core definitions for handling the kernel objects such as *udata*. This also decreases the number of dependencies and also the number of library calls. Figure 3 shows the same example using *virtio-rdma* architecture. Instead of sending the uverbs command to *OFED* kernel drivers, it does the hypercalls directly with the help of *virtio-rdma*. The unnecessary libraries have been eliminated, which saves up to 5 library calls per verb command.

3.4 RDMA Memory Mapping

The RDMA memory regions, like queue pairs, completion queues and work requests, are directly shared between the guest and the host, and the InfiniBand hardware is able to operate on them via DMA. This is the fundamental rule followed by both *Hyv* and *virtio-rdma* to accelerate the data path and off-load the CPU.

When using *Hyv* guest driver with Linux, the kernel will assign memory with contiguous virtual address but potentially non-contiguous regions in physical space, as shown in Figure 5a. The guest driver needs to parse the physical address and save the starting physical address, offset, and size of each contiguous region (chunk) in a mapping list. The list will be passed to the host driver, where each chunk will be translated back to the host physical address space and then mapped with the InfiniBand hardware. At the end, the backend driver holds a contiguous virtual address of the same memory region.

However, for OSv, the core memory allocator works differently as on Linux. The allocated memory is always contiguous in both virtual and physical address spaces (Figure 5b).



(a) Hyv (b) virtio-rdma

Figure 5: Comparison of memory mapping mechanisms between *Hyv* and *virtio-rdma*.

The memory translation is then made simpler: the guest driver takes care of the offsets in the first page and last page (all memory in between should be already contiguous and page aligned); it prepares the mapping list with physical memory of each chunk and pass them to the host; the backend driver will still map the guest memory into non-contiguous pages as it required.

The memory mapping process in *virtio-rdma* saves the effort of retrieving allocated pages of the the memory region, and finding contiguous pages by traversing and comparing page addresses.

3.5 Shared Memory Support

In this new design, *virtio-rdma* is capable for inter-host communication across the host through the hypercall path, it is also designed to support intra-host communication on the same host by supporting shared memory protocol. The VM and its application have no knowledge about the network topology, but VMs on the same host have the same *RDMA* device handle. Whether the communication should use shared memory or not is decided by comparing the device handles of peers.

The shared memory module in *virtio-rdma* is based on *ivshmem* [**?**], also known as *Nahanni*. It is implemented in the *virtio-rdma* driver, and it can be dynamically loaded when required. For example, when registering the *RDMA* memory, i.e. calling *ibv_reg_mr* verb API, the same memory region will be expose to be shared to other VMs on the same host by the shared memory module. Only if the communication peers has the same device handle, i.e. on the same host, the shared memory will be used for transmitting the data. The access permission is also protected by the *RDMA* communication rules, e.g. protect domain. For example, when calling *ibv_post_send* to post send to the VM on the same host,, we do not go futher with the *RDMA* stack, but rather use the implemented shared memory functions, which still follows the *RDMA* scenario: update the send request; tag the access permission of the send buffer in the queue pair to the other VM using the shared memory functions; update the completion queue by the frontend driver.

3.6 Support for socket

The purpose of supporting for *socket* in *virtio-rdma* is to accelerate the *socket* communication using the internel protocol, The *virtio-rdma* frontend driver needs to expose *socket* API to the guest application, and replace the *TCP* stack by virtual *RDMA* or shared memory. Two frameworks, *rsocket* [**?**] and *libvma* [**?**] have been evaluated and will be taken as the foundation of this work.

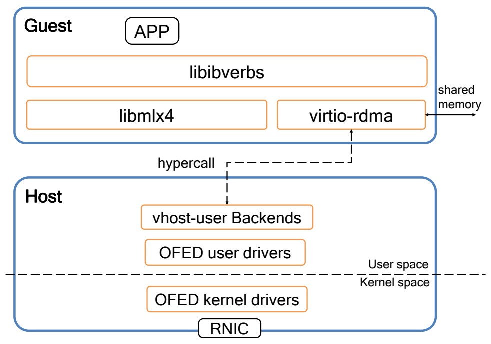


Figure 6: *Virtio-rdma* using *vhost-user* as the backend.

3.7 Improvement

One alternative and advanced way of doing the hypercalls in *virtio-rdma* is to compose hypercall message using the user context, which does not need the context switcher and OFED core definitions. However, this requires several changes on the host side, as shown in Figure 6: the backend driver should know about the user verbs, i.e. replacing the kernel verbs abstraction with user verbs API; the backend driver needs to work in user space. The key for this solution is *vhost-user* [13]. Converting the hypercalls with user verbs may simplify the design of *virtio-rdma* by omitting the user to kernel context switch and the OFED core definitions support. On the host side, it requires additional work to adjust vhost driver with *vhost-user* implementation.

4. HPC INTEGRATION

Our ultimate goal is to integrate *virtio-rdma* solution into a high performance cloud system, allowing for optimized network I/O performance. To boot a VM on a physical node in an HPC environment, the batch system requires certain adaptions and extensions. In this section, we describe the modification that needs to be carried out with resource manager and batch scheduler *Torque*, in order to run a job on a HPC system with *virtio-rdma* support.

Compared to traditional bare metal execution, a virtualized batch job's lifecyle is comprised the following:

- During the setup phase, the *prologue* [?] script generates metadata for customizing the VMs during boot by the help of *cloud-init* [15]. It also installs missing packages, creates the user account, and mounts shared file-systems available on the physical nodes. Further, the *prologue* script sets up the *virtio-rdma* and waits for the VMs to become available via SSH.

- Job execution is wrapped by another script that prepares the VM's environment variables for the applications. After preparing the VMs via SSH, the user job script is executed in the first virtual guest.

- In the tear down phase of a job's lifecycle, the *epilogue* [?] script cleans up all the instantiated VMs, including their *virtio-rdma* configuration.

The proposed workflow is completely transparent to the user, and carries out the same procedure for both jobs on VM and on bare metal node. Moreover, it skips the loading of extra modules, because in a virtualized environments applications images are packaged with all necessary dependencies.

5. CONCLUSION AND FUTURE WORK

In this paper, we present a design of a lightweight paravirtualized RDMA solution, called *virtio-rdma*, that implements a new frontend driver compatible with the *HyV* host driver with more advanced features, for example simple context switch and shared memory support. It is designed in a much simpler way than the original *HyV* architecture, due to the characteristics of *OSv* on the guest driver side. Our *virtio-rdma* approach supports *OSv* in its current state of development. However, we simplified the dependencies of *OFED* kernel modules and minimized support of *OFED* user libraries to provide API linkage to user application, allowing for future adaptation in any similar library OS *unikernels* or even Linux.

Furthermore, we proposed a concrete solution of adopting *virtio-rdma* in HPC environments with an extended *Torque*, allowing end-users to easily explore our para-virtualization solution without any prior domain knowledge.

Future work comprises the plan to evaluate and compare the I/O performance of the proposed implementation, on the extended virtualized HPC environments, to the traditional bare metal execution. ïż£

Acknowledgement

This work is supported by the MIKELANGELO project (grant agreement No. 645402), which is co-founded by the European Commission under the Information and Communication Technologies (ICT) theme of the H2020 framework programme (H20202-ICT-07-2014: Advanced Cloud Infrastructures and Services).

6. REFERENCES

[1] A. Ranadive and B. Davda, “Toward a Paravirtual vRDMA Device for VMware ESXi Guests,” pp. 1–18, 2015.

[2] J. Pfefferle, P. Stuedi, A. Trivedi, B. Metzler, I. Koltsidas, and T. R. Gross, “A Hybrid I/O Virtualization Framework for RDMA-capable Network Interfaces,” in *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE ’15. New York, NY, USA: ACM, 2015, pp. 17–30. [Online]. Available: http://doi.acm.org/10.1145/2731186.2731200

[3] “Torque Resource Manager,” http://www.adaptivecomputing.com/products/open-source/torque.

[4] “Introduction to InfiniBand,” http://www.mellanox.com/pdf/whitepapers/IB_Intro_WP_190.pdf.

[5] “RDMA over Converged Ethernet,” https://en.wikipedia.org/wiki/RDMA_over_Converged_Ethernet.

[6] “ RDMA Aware Networks Programming User Manual,” https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

[7] R. Bott, “RDMA Protocol verbs Specification,” *Igarss 2014*, no. 1, pp. 1–5, 2014.

[8] A. Kivity, D. Laor, G. Costa, and P. Enberg, "OSv — Optimizing the Operating System for Virtual Machines," *Proceedings of the 2014 USENIX Annual Technical Conference*, pp. 61–72, 2014.

[9] D. R. Engler, M. F. Kaashoek, and J. OâĂŹtoole, "Exokernel: An operating system architecture for application-level resource management," 1995, pp. 251–266.

[10] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," *SIGPLAN Not.*, vol. 48, no. 4, pp. 461–472, Mar. 2013. [Online]. Available: http://doi.acm.org/10.1145/2499368.2451167

[11] R. Russell, "virtio: Towards a De-Facto Standard For Virtual I / O Devices," *ACM SIGOPS Operating Systems Review*, vol. 42, pp. 95–103, 2008.

[12] J. Pfefferle and P. T. R. Gross, "vVerbs - A Paravirtual Subsystem for RDMA-capable Network Interfaces," no. April, 2014.

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