A Hybrid I/O Virtualization Framework for RDMA-capable Network Interfaces

Jonas Pfefferle*, Patrick Stuedi*, Animesh Trivedi*,
Bernard Metzler*, Ioannis Koltsidas* and Thomas R. Gross‡
IBM Research*, ETH Zuerich‡
{jpf,stu,atr,bmt,iko}@zurich.ibm.com, trg@inf.ethz.ch

Abstract

RDMA-capable interconnects, providing ultra-low latency and high-bandwidth, are increasingly being used in the context of distributed storage and data processing systems. However, the deployment of such systems in virtualized data centers is currently inhibited by the lack of a flexible and high-performance virtualization solution for RDMA network interfaces.

In this work, we present a hybrid virtualization architecture which builds upon the concept of separation of paths for control and data operations available in RDMA. With hybrid virtualization, RDMA control operations are virtualized using hypervisor involvement, while data operations are set up to bypass the hypervisor completely. We describe HyV (Hybrid Virtualization), a virtualization framework for RDMA devices implementing such a hybrid architecture. In the paper, we provide a detailed evaluation of HyV for different RDMA technologies and operations. We further demonstrate the advantages of HyV in the context of a real distributed system by running RAMCloud on a set of HyV-enabled virtual machines deployed across a 6-node RDMA cluster. All of the performance results we obtained illustrate that hybrid virtualization enables bare-metal RDMA performance inside virtual machines while retaining the flexibility typically associated with paravirtualization.

1. Introduction

RDMA-capable interconnects like Infiniband or iWARP are increasingly being considered in deployments of large distributed data processing systems. Examples of this sort are RAMCloud [22], FaRM [5] and Pilaf [18]. RDMA networks provide ultra-low latency and high-bandwidth – two properties that are of highest interest when processing large data sets under time constraints.

As of today, all of the existing systems we are aware of that use RDMA networks in larger deployments do so in a strictly bare-metal environment. This is unfortunate since many of the systems would benefit from the advantages of virtualization (higher utilization of physical resources due to hardware multiplexing, flexibility due to virtual machine migration, snapshotting, etc.) [3, 12].

Currently, virtualization support for RDMA network interfaces is available in the form of single-root I/O virtualization (SR-IOV) [25]. SR-IOV delivers close to bare-metal performance in virtualized environments but lacks some of the flexibility aspects one typically associates with virtualization. For instance, SR-IOV requires device and platform support, which, compared to software solutions, has higher development costs and is harder to maintain. Additionally, setting up SR-IOV is inherently static which complicates virtual machine migration later on (see Section 2.2).

In this paper, we argue that the very nature of RDMA's separation of control and data path allows for virtualizing RDMA capable NICs entirely in software while still achieving bare-metal performance. We introduce the notion of hybrid virtualization, a I/O virtualization scheme for RDMA interconnects. With hybrid virtualization, virtual RDMA devices running in the guest interact with the hypervisor using a paravirtual interface solely for control operations such as when creating network resources like queue pairs or completion queues. These network resources are then mapped into the guest virtual machine for direct access. Consequently, any subsequent access to these resources during data transmission and reception bypasses the hypervisor completely.

We present HyV, a virtualization framework for RDMA-capable network interfaces implementing such a hybrid architecture for the Linux kernel virtual machine (KVM). We show that HyV is able to achieve highest performance while still retaining maximum flexibility. As an example, HyV permits applications in one virtual machine to read or write

VEE '15, March 14–15, 2015, Istanbul, Turkey.

http://dx.doi.org/10.1145/2731186.2731200

memory from remote virtual machines on different physical hosts within 1-2 microseconds.

By using the HyV framework, virtual RDMA devices can be implemented very similar to implementing paravirtual devices. Thereby, the key challenge in HyV was (1) to separate the hardware-dependent part from the hardware-independent part and (2) to facilitate and ease the effort for vendors to implement their own virtual RDMA NICs with very little effort. The paper presents the design and implementation of HyV and discusses three specific RDMA devices which we virtualized using HyV.

To demonstrate the advantages of HyV for real applications, we have deployed RAMCloud – a DRAM-based distributed key-value store – on a set of HyV-enabled virtual machines. Our measurements show that by leveraging HyV we can achieve native throughput and latency on the cluster with no change to the application.

In summary, this paper's contributions include (1) a hybrid I/O architecture to virtualize RDMA network devices, (2) the design and implementation of HyV as a generic framework for virtual RDMA devices, and (3) the demonstration of how to use HyV to achieve bare-metal latency and throughput between virtual machines interconnected with a RDMA capable network.

2. Background

In this section we provide the relevant background on RDMA networking and network virtualization.

2.1 Remote Direct Memory Access (RDMA)

RDMA capable network interface controllers or RNICs offer high-bandwidth, ultra low-latency network operations for accessing data in remote memories. The performance advantages of RDMA are mainly achieved by separating the *control path* from the *data path* while completely eliminating the OS/CPU involvement from the latter. This separation of paths frees network I/O from overheads otherwise associated with resource management, multiplexing, scheduling etc. [6]. Instead, network and application resources are pre-allocated and registered with the RNIC on the control path. A control path to the RNIC consists of calls to setup resources and context in the device that – during data path operations – enable moving data in a zero-copy fashion between the network and an application buffer. This separation philosophy is also reflected in the way applications interact with the RNIC. Applications typically identify and create necessary resources upfront and outside of performance critical sections, but benefit from fast RDMA data operations when performance really matters. In comparison, the traditional BSD socket interface has intertwined control and data paths where resources are allocated, associated, used and then released all as part network I/O operations.

Thus, the critical aspect of RNIC's is that the control paths are used to directly map device provided, connection-specific, structures into an applications address space. For example, RDMA gives each application its own private and virtual network interface consisting of transmission (TX) and reception (RX) work queues called *queue pairs* (QP). This approach eliminates software and protection overheads, however, it also entangles application code with device details that are usually encapsulated by an OS device driver. To mitigate this burden and ensure portability a device's specific memory layout and protocol for interaction is encapsulated by a vendor provided user-level library. The library exposes a standardized RDMA interface (verbs) to the application while managing and interacting with mapped structures unique to a particular device [9, 11]. To issue a data transfer, applications use the verbs interface to asynchronously post I/O requests on the QP. These I/O requests contain pointers to application buffers that are accessed by the RNIC through DMA. I/O completion events are put on a separate notification queue which applications can poll or block on.

2.2 Network I/O Virtualization

Virtualization enables efficient hardware utilization by multiplexing server hardware among virtual machines (VMs). As virtual machines can be created, snapshotted, replicated, and resumed very quickly, they give great flexibility in resource planning, provisioning and administration to handle workload spikes. Despite many advantages, one of the key challenges in virtualization is efficient network I/O virtualization. The two most prominently used techniques for network I/O virtualization are (a) direct device assignment and (b) paravirtualization. Direct device assignment can be achieved by passthrough or hardware assisted virtualization (e.g. PCIe SR-IOV). In both cases a dedicated NIC instance is exclusively assigned to a virtual machine. While this mode avoids any involvement of the hypervisor, it lacks flexibility. For example, setting up direct device assignment is typically a static task where the number of NIC instances need to be configured upfront and cannot be changed thereafter. This requires careful consideration to not waste resources on the hardware for unused virtual interfaces. Further, direct device assignment needs special platform support for memory translation and complicates VM live migration [24].

In contrast to hardware-assisted virtualization, a paravirtualized network stack offers great flexibility at lower performance. It uses a split driver model with a frontend driver in the virtualization aware guest OS and a backend driver on the host. Frontend and backend driver interact over a dedicated communication channel[7]. With paravirtualization, the software network state is maintained in the hypervisor which simplifies migrating and checkpointing of the network state. These flexibility aspects of paravirtualization are the main reason why paravirtualization is widely used in todays virtualization environment. The downside of paravirtualization is that crossing the guest/hypervisor boundaries imposes some overhead during data path operations.

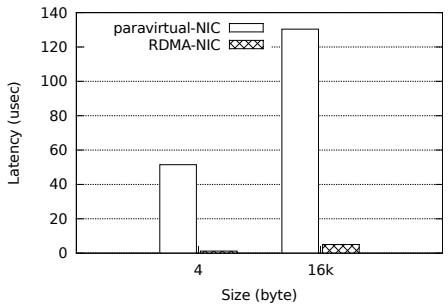


Figure 1: Performance gap between paravirtualized socket/tcp, and bare-metal verbs/rdma

In the following, we discuss our hybrid virtualization scheme tailored to RDMA-capable interfaces. Hybrid virtualization inherits some of the flexibility advantages of paravirtualization, but at the same time, achieves a performance comparable to hardware-assisted virtualization.

3. Hybrid RDMA Virtualization Architecture

Paravirtualized network devices use a single channel to communicate between the guest and the host operating system. Both data and control transfer funnel through the channel and many layers of abstractions, requiring data copies, hypervisor accesses, scheduling, network buffer allocation and management etc. This design originates from the fact that traditional network controllers do not implement a sufficient form of isolation and context to allow network resources to be multiplexed across hypervisor/guest boundaries. A trusted entity (i.e. hypervisor) has to process data and control operations in software before transfer to or from untrusted guests (Figure 2a). This is in a sharp contrast to RDMA operations where the data and control paths are completely separated even at the device interface level (Figure 2b). The separation helps to deliver high-performance with isolation to the applications. Consequently, there exists a large performance gap between RDMA performance and the performance of a paravirtualized network stack (Figure 1). To enable RDMA performance inside virtual machines, it will therefore be absolutely key to continue separating the control from the data path. The traditional paravirtualized architecture is not sufficient since it does not include the notion of path separation, and as such cannot deliver the full performance advantages of RDMA.

Design: A natural way to extend RDMA's philosophy of separation of paths into virtualization is to virtualize RDMA network devices at the function interface of RDMA verbs, instead of at the level of the PCI device. We define a hybrid virtualization architecture for RDMA interfaces in which verbs-level control operations are virtualized in a paravirtual fashion, while data operations are executed on guest-private data channels (Figure 2c).

3.1 Paravirtualized Control Path

On the control path, network resources are mapped to and from the RDMA controller into the application memory inside a VM. With hybrid virtualization, these network resources are created using a frontend/backend driver pair. Control operations issued from an RDMA-enabled application – such as `create_qp()` to create a queue pair, or `create_cq()` to create a completion notification queue – are executed by performing a system call into the frontend driver running in the guest. This mechanism is similar to the one used for control operations in the native RDMA stack. After having intercepted the call, the frontend driver forwards the control operation to the backend driver using a paravirtual communication channel. The backend driver, if needed, enforces the isolation, security and resource limits for VMs by modifying/filtering operations before forwarding them to the *unmodified* host device driver. After approval from the host OS, the host device driver creates the requested network resources on the RNIC. Finally, the backend driver maps the newly created resource (e.g., queue pair) into the address space of the guest RDMA application. Application resources such as memory buffers are also registered (*reg_mr()*) with the RNIC using the split drivers.

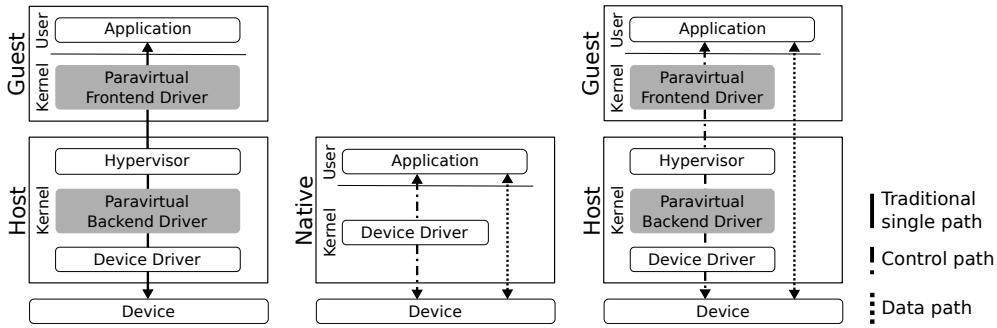
In order for hybrid virtualization to work together with unmodified host RDMA drivers, some control operations – in particular the ones that interact with the memory management subsystem – require modifications in the host. Specifically, extra memory translations are needed in two directions:

1. **Top Down:** Translating guest virtual memory areas into the host address space such that they can be used with the host device driver and RNIC.
2. **Bottom Up:** Mapping host physical memory areas to guest virtual memory such that it can be used by a guest application to directly access the RNIC.

These two kinds of mappings cover scenarios for both data path setup and user memory registration. However, the implementation depends on how the hypervisor handles virtual machine memory as well as on the interface to the host device driver, details of which are discussed in the Section 4.2.

3.2 Direct Data I/O Path

The data path consists of posting I/O requests and reaping completion notifications from the RNIC. With hybrid virtualization, this is achieved by directly reading and writing memory-mapped hardware queues from within the guest. The RDMA application in the guest OS prepares an I/O request and writes it in the queue pair using the `post_send()` or `post_recv()` verbs call. Since the queue pair is mapped into guest memory, the unmodified user-space device-driver can be used for posting I/O requests.



(a) Paravirtualization (b) RDMA (c) Hybrid virtualization

Figure 2: Virtualization architectures

A transmission work request triggers the RNIC to prepare and transmit data from the pre-registered application buffer inside the guest OS in a zero-copy manner. For incoming packets, the destination memory is resolved by the RDMA controller after RDMA protocol processing and data is directly DMA'ed into the final application buffer. Since network and packet processing, multiplexing etc. happens in the RDMA controller, the hypervisor (as well as the guest/host OS) is completely eliminated from the fast data path. Upon the I/O completion, the RNIC generates a work completion (WC) element and puts that into the completion queue. The application can poll (`poll_cq()`) or block (`get_cq_event()`) on this queue to receive completion notification. The blocking mode requires further support from the underlying virtualization framework to raise events for notifications. Event notification is discussed in more detail in Section 4.

3.3 Advantages

Hybrid virtualization has advantages over both direct assignment based virtualization and paravirtualization. Clearly, when compared to paravirtualization, the advantage of hybrid virtualization is the improved performance due to less hypervisor involvement. By eliminating the guest OS and the hypervisor from the fast data path, the hybrid virtualization architecture can deliver a performance that is very close to the network hardware limits.

On the other hand, when compared to direct devices assignment, the advantage of hybrid virtualization is that it puts the hypervisor back in charge of network resource creation, management and accounting on the RNIC. Now these operations can be implemented by taking any necessary operating system and virtual machine state into consideration. Consequently, the hypervisor can enforce various firewall, quota, and isolation rules while creating network resources on the RNIC. The hybrid architecture is also more resource efficient. Instead of statically partitioning network resources between virtualized instances, in the hybrid architecture these resource can be assigned on-demand to VMs. For example, SR-IOV with IOMMU requires allocation and pinning of all memory of the guest VM for DMA. With hybrid virtualization, only application buffers which are involved in RDMA are pinned.

4. Implementation

We developed HyV, a proof-of-concept implementation of the hybrid virtualization architecture. HyV offers an easy to use, flexible, and generic RDMA virtualization framework implemented for the Linux kernel virtual machine (KVM) as the hypervisor [14]. The system runs on the x86-64 architecture, however, our source code does not have any architecture dependencies and, although not tested, should run on other platforms as well. The ultimate goal of HyV is to provide a virtualization solution for RDMA that fulfills the following requirements:

1. *Direct hardware access:* For full performance, applications inside a VM should be able to directly access the RDMA network interface.
2. *Standardization and extensibility:* The framework should make it easy for vendors of RDMA hardware to get their devices virtualized.
3. *Unmodified kernel and userspace components:* Existing native RDMA applications should run unmodified in a HyV-enabled virtual machine.

Requirement (1) is achieved by implementing the hybrid virtualization architecture as discussed in the previous section. Requirements (2) and (3) are achieved through a modular implementation of HyV. In the following, we first provide a brief overview of the software architecture of HyV, and then discuss several aspects in more detail.

4.1 Overview

HyV is implemented as a plugin component for the Linux OFED RDMA software stack. OpenFabrics is an effort by different vendors to integrate RDMA interconnect technologies and provide a standard RDMA verbs programming

interface [19]. Their OpenFabrics Enterprise Distribution (OFED) is a widely used RDMA software stack which spans both user- and kernel-space. User-space applications link against the *libibverbs* library which forwards control verbs operations to the *OFED core* in the kernel; from there they are further forwarded to a device-specific kernel-driver. Data verb calls are forwarded to a device-specific user-space driver where network resources are directly accessed, bypassing the operating system. By programming against the OFED provider API, vendors can enable their devices with a user- and kernel-space driver.

Figure 3 shows the architecture of HyV as it is embedded in the OFED stack. HyV allows the unmodified OFED stack to be used by both guest and host OS. Applications inside a virtual machine will use the regular components comprising of libibverbs, OFED core and user-space device drivers. Instead of running the real kernel-level provider, however, HyV requires a virtual provider to be loaded inside the guest. The virtual provider interacts with the HyV frontend to relay verbs control operations to the hypervisor. At the hypervisor, a HyV backend receives the control operations and feeds them into the OFED core at the host.

The virtual provider loaded in the guest is device-specific and is the only component in HyV that needs to be implemented to virtualize a given RDMA network interface. A virtual provider inside the guest is necessary because setting up the memory-mapped data path during control operations is device-specific. Vendors of RDMA hardware develop a virtual provider by implementing the standard OFED function calls that are otherwise implemented by their real kernel-device driver. For instance, by implementing the `create_qp()` interface, a vendor is given the opportunity to implement a device specific mapping of the QP resource. The HyV framework provides a set of API functions which makes the development of a virtual provider for a given hardware easier. As an example, HyV provides generic functions to map and unmap host memory into the guest address space. Later, we discuss the implementation of virtual providers for one Infiniband and two iWARP RDMA devices.

As discussed earlier and also illustrated in Figure 3, only control operations are relayed through the hypervisor (via virtual provider), data operations are executed directly on the RDMA hardware using the previously established mapped resources in the guest.

Control Path: We use the virtio paravirtualization framework as a basic mechanism to pass control operations from the guest to the hypervisor [29]. virtio uses a queue abstraction for the transport layer, called virtqueue, which allows buffers to be exchanged between guest and host. Hypercalls are used to notify the host of newly available buffers that are ready to be transmitted, and injected interrupts are used to notify the guest of newly received data buffers. On x86, notifications require VM exits and therefore add some extra

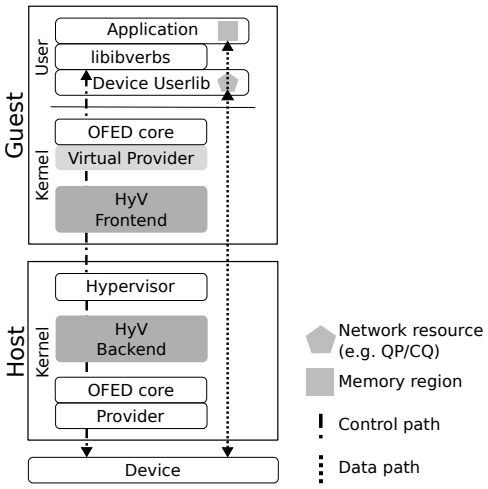


Figure 3: HyV architecture

latency. We will evaluate the effect of interrupt injection and hypercall VM exits in Section 5.

Data Path Work Completions: Remember that RDMA offers two ways of notifying the application about a completed RDMA operation, polling and blocking. With HyV, no hypervisor interactions are necessary when polling is used. In the case of blocking, the situation is different. Here, the application is put to sleep and woken up as soon as work completions are put on the CQ. One possible way to implement such a notification mechanism is to forward callback operations using the virtio framework. However, we found the virtio framework to be inefficient at handling small buffer events (e.g., completion events of 4 bytes). Namely, virtio uses extra work queues to notify a guest of a consumed buffer which adds latency to the performance critical event path. Consequently, we decided to use a shared ring-buffer instead of virtqueue buffers to forward events to the guest. The ring-buffer is an implementation of a single consumer/producer queue [15] which is protected by guest and host private locks to allow for multiple consumers/producers. These additional locks are necessary because Linux defers interrupt processing to software interrupt handlers which can run in parallel on different CPU cores [17].

4.2 Resource Mapping

The fundamental task of RDMA control operations like `create_qp()` or `reg_mr()` is to map resources (e.g., queue pair, or user memory) either from the host into the application inside the virtual machine, or, from the guest application to the host.

Our observation with RDMA devices is that there are two basic schemes used to setup such a resource mapping: *bottom up* - the application reserves virtual memory in the form of a memory map call to a special file and the kernel device driver backs them by physical pages (e.g. DMA-

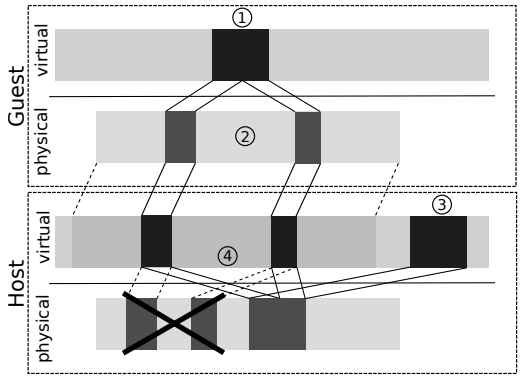


Figure 4: Bottom up memory mapping

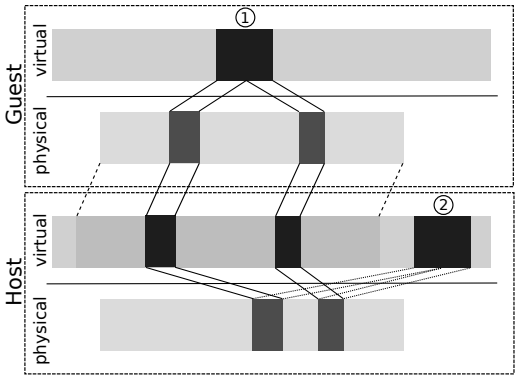


Figure 5: Top down memory mapping

enabled pages or PCI I/O memory); *top down* - memory is allocated in the application and pinned by the kernel device driver to enable DMA transfers.

It turned out that implementing such a mapping functionality in case of virtualization is challenging due to several assumptions made by the OFED host drivers. In the following we describe the challenges and explain how we addressed them in HyV. Note that we cannot use an IOMMU to perform these mappings because we share a single device across virtual machines with different physical address spaces. Devices can initiate DMAs to any address spaces at any time.

Bottom up: The provider on the host expects a contiguous (host) virtual memory region, ready to back it with physical pages. Unfortunately, a contiguous virtual memory region in the guest might not be contiguous in the host virtual memory (HVM). This is because in KVM virtual machines are essentially regular Linux processes, each of which with its own virtual address space.

To overcome this problem in HyV, we remap guest physical memory (GPM) to the actual physical pages of the network resource. The various steps of this process are illustrated in Figure 4. First, the application performs the mapping as is (step 1), which ends up in the virtual provider. From there the call is forwarded to the HyV frontend where the reserved virtual memory region is backed by physical pages (step 2). Subsequently, the HyV frontend passes the list of physical pages and the initial mapping parameters to the HyV backend where the pages are translated to the corresponding virtual memory regions on the host (offset calculation). Using the initial mapping parameters, the HyV backend creates a mapping into host virtual memory (HVM) to extract the physical pages of the network resources by walking the page table (step 3). Finally, the backend remaps the virtual memory regions that correspond to the mapping in the guest, to the extracted pages of the actual network resource (step 4). The HVM mapping is not needed anymore and can be destroyed.

Top down: Here, the provider also expects a contiguous (host) virtual memory region, but unlike in *bottom up* it does not back the region itself but relies on the operating systems to do so. Thus, the provider only pins the memory to ensure that the pages can be used for DMA transfers and extracts them for installation on the device. Unfortunately, in the virtualized environment we have the same problem as in *bottom up* that the virtual memory region from the guest is not contiguous on the host.

Again, in HyV we implement a memory remapping between the guest and host. The various steps are illustrated in Figure 5. This time we have to use the underlying physical pages that back the virtual memory region in the guest (step 1). Thereafter, we pin the physical pages in both the guest and the host in order to create a new contiguous mapping into HVM (step 2). This mapping can then be used by the provider to extract and pin the physical pages as described previously. After registration the HVM mapping can be destroyed. HyV does not need to perform a remapping into contiguous HVM if the memory region is already contiguous, i.e. contiguous in GPM. Currently, we do not fully leverage this fact, but avoid remapping if the memory region fits inside a single page.

4.3 RDMA Devices Supported by HyV

A virtual provider for a specific RDMA network interface needs to implement various handlers for verb control operations like `create_qp()` or `create_cq()`. These handlers are invoked by the OFED runtime everytime an application issues a corresponding verb control operation. A virtual provider can implement a given handler in two ways. Either – for instance when implementing the `reg_mr()` handler – it can relay the operation directly to the real kernel-level provider by using the matching relay API function available in HyV. Or – for instance when implementing the `create_qp()` handler – a provider may use the mapping API functions available in HyV to map network resources into the guest in a device specific way. Table 1 shows the most important API functions available in HyV for implementing virtual providers.

To emphasize the framework aspect of HyV and to show that it is not limited to a particular interconnect technology

API functions	Description
`hyv_reg_mr()`	forward `reg_mr()` operation
`hyv_create_qp()`	forward `create_qp()` operation
`hyv_mmap_gvirt()`	top down memory mapping
`hyv_mmap_hphys()`	bottom up memory mapping

Table 1: Most prominent HyV API for implementing virtual providers

we implemented virtual providers for both Infiniband and iWARP devices. With these providers we are able to support the following devices:

Mellanox ConnectX-1/2/3 VPI/Pro: The ConnectX device supports Infiniband [9] and RDMA over converged Ethernet (RoCE) [10]. Its driver allocates resources in the user-space driver and requires a *top-down* mapping in the virtual provider. Additionally, its driver requires a user accessible region (UAR), e.g. used for doorbells. The UAR is allocated in a *bottom up* manner and can be easily mapped by the provided mapping API.

Chelsio Terminator 4/5: This is a 10GbE NIC with a fully offloaded RDMA iWARP [27] engine. iWARP implements RDMA functionality on top of a TCP/IP transport. The T4/5 device allocates network resource in a *bottom up* manner. Therefore, the resource mappings for T4/5 could easily be implemented using the aforementioned mapping API available in HyV.

SoftiWARP: This device is a software implementation of iWARP on top of kernel TCP sockets [30]. Applications using SoftiWARP benefit from zero-copy data transmission. However, data operations do involve the kernel and require a system call to kick of processing. This system call is implemented by a special `post_send()` operation to the kernel. The virtual provider for SoftiWARP therefore has to virtualize the `post_send()` operation which is done by forwarding the operations the same way as control operations. Unfortunately, this also adds extra latency during data operations. We are going to evaluate the latency overhead of virtualized SoftiWARP in Section 5.

5. Evaluation

In this section we evaluate HyV at the level of raw RDMA operations. We demonstrate HyV in two different experimental setups.

Experimental setup (A): A 6-node Infiniband cluster where each machine contains a dual Intel Xeon E5-2650 v2 (2.6Ghz, 20MB Cache) CPU and 160GB DDR3 RAM. The machines are connected using a dual port Mellanox ConnectX-3 VPI 56Gb/s FDR Infiniband RNICs. We use two back-to-back connected interfaces on different physical machines for microbenchmarks. Experiments in Section 6 use all machines with both ports connected through a Mellanox SX6036 Switch. On these systems we use the KVM hypervisor of a 3.13.11 vanilla Linux kernel with a QEMU frontend [4]. All virtual machines are configured with six virtual CPUs and 16GB memory and they run a 3.13.11 vanilla Linux kernel with *KVM_GUEST* kernel configuration enabled. We compare three configurations: native, SR-IOV and with HyV.

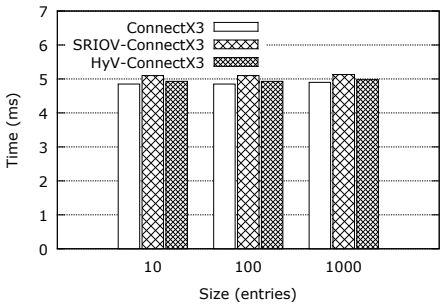


Figure 6: Create QP

Experimental setup (B): Two machines with dual Intel Xeon E5-2690 (2.9Ghz, 20MB Cache) CPU and 96GB DDR3 RAM. The machines are connected using Chelsio T420-CR 10GbE RNICs with jumbo frames (9K MTU). The NICs are connected through an IBM G8264 Switch. Kernel versions and configuration are the same as above.

5.1 Control Path

In the following, we quantify the performance overhead of control operations in HyV which is required to enable complete bypass on the data path. All experiments are performed using experimental setup (A).

The `create_qp()` verbs call creates a queue pair and installs it on the RNIC for direct access. In this test we create QPs that can hold 10, 100 and 1000 work request entries and compare the virtual device's control path to the control path of the native device. We measure the performance by counting the number of transactions (creating and destructing QPs) over a period of 10 seconds. As can be observed from Figure 6, HyV shows slightly lower performance than native, as it has to double pin and remap the QP memory (*top-down* mapping, cf. Section 4). This shows that creation time is actually dominated by installing the resource on the device. SR-IOV has similar but slightly worse performance than HyV. We suspect more complex communication and resource management on the device when dealing with virtual functions (VF) to account for lower performance on the control path.

The `reg_mr()` verbs call registers a memory region for later use, causing the NIC driver to pin the memory region and install its physical pages onto the NIC. Figure 7 shows the registration cost for memory regions with sizes from 1KB to 256MB. Memory allocation and population cost is not included in the measurement. For sizes smaller or equal to 64KB, the memory registration time is more or

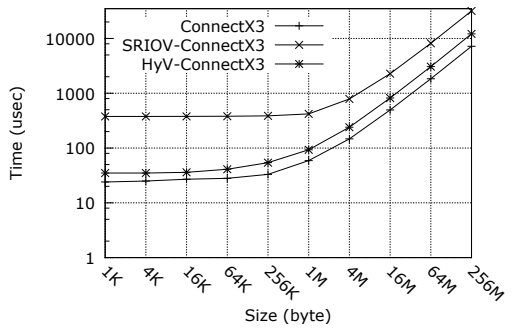


Figure 7: Memory Registration

less constant and is determined by the cost of processing the system call and manipulating the process address space. Except for SR-IOV where, as above, registering seems to be limited by communication overhead with the device. For larger sizes, the registration cost is dominated by the number of pages being registered. Registering a MR with HyV-ConnectX3 takes approximately 2 times as long as with native ConnectX3. For small sizes this overhead mainly comes from forwarding the command over the paravirtual control path. For larger transfer sizes the overhead can be accounted to the double-pinning of pages (guest and host) as well as to the cost of remapping memory. SR-IOV takes approximately 4 times as long as compared to native ConnectX3. To get a better understanding if this is a general PCI passthrough/SR-IOV issue we conducted the same experiments on a passthrough-T4 device (not shown) and observed that this setup only shows minor performance overhead on the control path. We plan to further investigate the exact cause that limits the ConnectX-3 SR-IOV control path performance.

A memory region (MR) can be unregistered with the `dereg_mr()` verbs, which unpins the memory and removes the physical page list from the NIC. We have also measured the cost of unregistering memory regions, but we omit discussing these results in detail as they were very similar to the results we got for memory registration.

5.2 I/O Performance

One key metric of interest is the raw I/O performance of HyV in terms of RDMA data operations. In the following, we look at throughput and latency of RDMA operations in HyV, and compare the results with the performance obtained natively and with passthrough. The benchmarks are performed either VM-to-VM or host-to-host.

Latency: To measure latency we use RDMA read operations with 4B and 16KB message size respectively. The reported numbers are round-trip times, i.e. work completions of read operations are generated after the remotely read memory has been stored locally. To determine the completion of a RDMA operation we use CQ polling. Experiments lasts 10 seconds and numbers reported are the average over 5 runs.

As can be observed from Figure 8a, the three different configurations, native, SR-IOV and HyV using experimental setup (A) show very similar performance for both 4B and 16KB message size. This is because in all three cases, the network resources (queue pairs, completion queues) are being accessed directly from the application in the guest, with no operating system or hypervisor involvement.

Figure 8c compares read latency of 4B and 16KB message size for the native and HyV-enabled T4 device when polling for completions using experimental setup (B). As seen in the previous Section, due to complete bypass of the hypervisor HyV-T4 is able to achieve equal to native latencies.

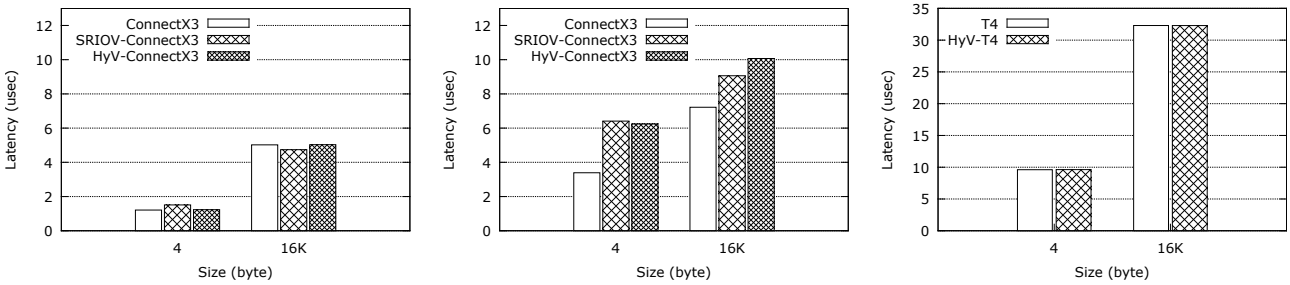
Throughput: To measure throughput we use RDMA write operations with message sizes from 1B to 64KB. The test starts with 10 outstanding work requests and posts a new work request each time a request has completed. The intention behind the small number of outstanding work requests is to capture a potential overhead introduced by the virtual environment. Benchmarks in Figure 9a are performed on experimental setup (A). We observe that HyV does not impose any overhead in terms of throughput. Due to hypervisor bypassing, passthrough and HyV show equal performance compared to their native counterpart. In each of the cases linespeed is reached at 4KB message size.

Throughput tests using experimental setup (B) with HyV-T4 (not shown) confirmed the above results. T4 is able to reach linespeed at 2KB transfer size as native (10 outstanding).

5.3 Virtualization Overheads

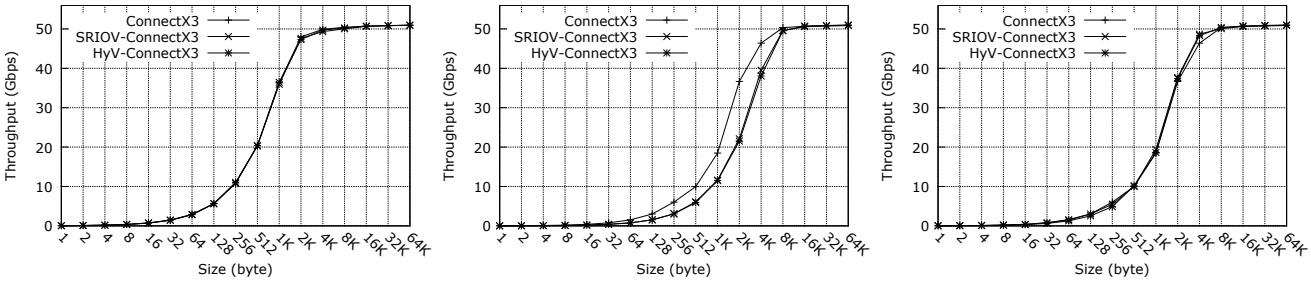
VM exits are one of the main sources of overhead in I/O virtualization in general. With HyV, VM exits occur as part of forwarding control operations or when using completion events. Fortunately, RDMA semantics allow completion event interrupts per operation, in contrast to traditional networking where in the worst case, interrupts are generated per packet.

Completion Events: Figure 8b shows the RDMA read latency with 4B and 16KB message sizes for the case where blocking is used instead of CQ polling on experimental setup (A). As can be seen, both SR-IOV and HyV show equal latency for 4B message size, but their latencies are higher compared to the latency in the native setup. For 16KB message size, HyV shows slightly higher latency than SR-IOV. The performance overhead for completion events can be explained by the need for an emulated advanced programmable interrupt controller (APIC) to allow interrupt injection with the current x86 architecture [8]. This introduces VM exits to inject interrupts into the VM on each completion event (see Section 4). However, as expected, the interrupt injection latency of 3μs for 4B remains constant with increasing



(a) Infiniband – Polling (b) Infiniband – Events (c) iWARP – polling

Figure 8: RDMA Read Latency



(a) Polling – 10 outstanding (b) Events – 10 outstanding (c) Events – 18 outstanding

Figure 9: RDMA Write Throughput on Infiniband

message size and thus, becomes negligible for large message sizes.

We further investigate the effect of CQ blocking on RDMA write throughput. Figure 9b-c shows write throughput for the two different batch sizes. In a configuration with just 10 outstanding work request, ConnectX3 reaches linespeed at 8KB. The performance gap for up to 8KB between the native and virtualized environment is due to interrupt injection overhead, cf. above. As we increase the number of outstanding work requests to 18, the overhead is amortized.

Memory footprint: The size of the memory footprint is an important metric in a virtualized environment and directly affects the level of consolidation that can be achieved. We ran an experiment where we register a memory region of 1MB and count the the number of pinned pages in the system. What we observed is that with the passthrough configuration the entire virtual machine memory was locked and held in the host RAM. In particular, we noticed that the virtual machine resident set size was 16GB, which is the maximum RAM the VM was permitted to use in the given configuration. The reason for this behavior is that passthrough requires locking of the complete guest physical memory to support direct DMA anywhere into the VM memory.

For the same experiment, but using a HyV-enabled VM, we observed only 1MB of memory being pinned and a resident set size of 205MB (guest application + kernel). HyV allows for a more efficient memory usage because all the RDMA memory (which is DMA'able) is explicitly registered through the hypervisor. Consequently, HyV has a much lower memory footprint than passthrough, and thereby allows packing more VMs on to a single physical machine.

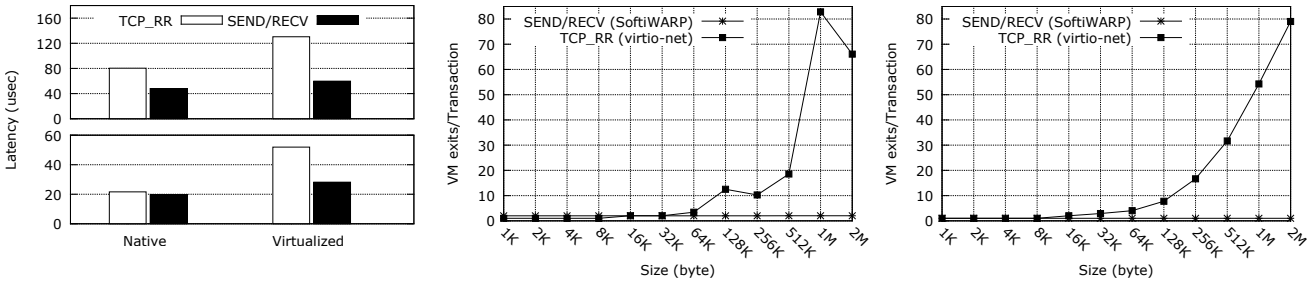
CPU load: CPU load is also an important metric to be considered in the context of I/O virtualization. Table 2 shows the host CPU load during RDMA operations (initiating side) for different configurations (100% is equivalent to 1 loaded core).

All configurations fully load a single CPU if RDMA operation are used in combination with polling. When using events, the CPU load is generally much lower since the application is temporarily put to sleep (as opposed to executing a busy wait polling loop). However, events lead to interrupt injections in a virtualized environment and therefore increase the CPU load for both passthrough and HyV.

5.4 Software Devices

In the following we show that VMs can directly benefit from HyV-enabled software RDMA devices where hardware is not available.

In Figure 10a we compare native and virtualized TCP request/response performance against Software RDMA send/



(a) 4B (bottom) and 16KB (top) message sizes (b) Interrupt Injections (c) Hypercalls

Figure 10: Send/receive on Software RDMA compared to TCP request/response

	read latency		write throughput	
	polling	events	polling	events
ConnectX3	100%	4%	100%	46%
sriov-ConnectX3	100%	35%	100%	96%
HyV-ConnectX3	100%	22%	100%	98%

Table 2: CPU load

Device	**Provider**	**Virtual provider**
ConnectX1/2/3	9165	357
SoftiWARP	7923	279
Chelsio T4/5	11483	536

Table 3: Virtual provider lines of code

receive on experimental setup (B). RDMA send/receive has similar semantics to socket `write()/read()`. All configurations run on a Chelsio T4 (RDMA offloading disabled) NIC. For virtualized TCP we use the virtio-net device with a vhost kernel backend. As software RDMA device we use SoftiWARP a iWARP implementation on top of kernel TCP sockets. HyV is used to virtualize the SoftiWARP device.

Our results show that SoftiWARP is capable of delivering lower latencies than TCP sockets. In the virtualized setup this performance benefit over TCP becomes more significant as HyV-SoftiWARP only introduces an overhead of approximately 10μs. In contrast virtio-net is up to 2.5 times slower than native TCP and, thus, can be outperformed by HyV by a factor of 2. This can be explained by additional copies in virtio-net and network stack traversals on both guest and host. In contrast, HyV offloads the TCP/IP processing onto the host.

We show interrupt injection exits in Figure 10b, hypercall exits in Figure 10c. HyV only requires two VM exits for interrupt injection per transaction, i.e. one for each operation (send/receive), independent of the transfer size. Moreover, HyV-SoftiWARP requires only one hypercall exit per operation. In contrast, the packet-based nature of virtio-net results in much larger numbers of VM exits. Note that Virtio-net does implement a certain level of batching to mitigate this problem, nevertheless the interrupt rate and VM exits is much higher compared to HyV.

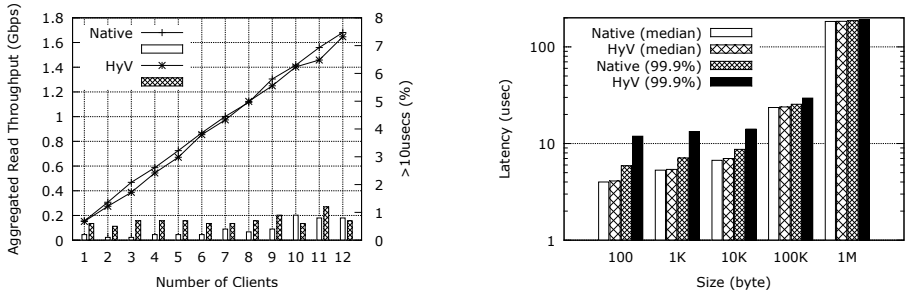
These results show the direct benefit of having Software RDMA devices in a virtualized environment. While interrupt injection and hypercalls add up to a large amount of VM exits respectively computation overhead on a traditional paravirtual NIC, HyV can totally avoid interrupt exits on data operations if polling is used, and even when using events only one VM exit per operation is required.

5.5 Virtualization Effort

As discussed, to virtualize a device with HyV, a vendor has to write a virtual provider. Table 3 shows lines of code for the original provider (running in the host) and their virtual counterpart for Mellanox ConnectX1/2/3, SoftiWARP and Chelsio T4/5. In all cases, the code size of the virtual providers is less than 5 percent of the original provider. Due to the framework API available in HyV, the virtual providers are easy to implement and do not require a deep understanding of the original provider. For example, it took us only 3 days to implement a virtual provider for the Chelsio T4/5 RDMA NIC.

6. Application: RAMCloud

RAMCloud [23] is a distributed DRAM-based key-value store. Durability and availability are provided by fast failure recovery from disks, instead of main-memory replication, due to cost and energy [21]. RAMCloud is built for large scale and low latency. The latter is achieved by performing RPCs over RDMA. This allows, for example, to execute a 100B read in under 5μs. RAMCloud's cluster architecture consists of three components: master and backup, coordinator and client. A storage server runs a master which holds all the data in DRAM and, typically, a backup of the data on disk. The coordinator is a centralized service managing metadata and cluster membership. Clients perform RPCs to the coordinator and to masters. In our experiments we run masters without backup, as our focus is on network I/O. We evaluate RAMCloud on experimental setup (A), cf. Section



(a) Aggregated system read throughput

(b) Single client read latency

Figure 11: RAMCloud deployed in virtual machines using HyV

5, both in a native deployment and virtualized with HyV. Two aspects of the system are measured: aggregated read bandwidth and single client latency.

Aggregated Read Bandwidth: In this experiment we run 12 masters and 1 coordinator co-located on 6-machines leveraging all 12 IB ports available. Each master is initialized with one table holding one entry with a 30B key and 100B payload. Clients perform random reads across all servers. We run clients co-located to the masters on all 6-machines with increasing number to show the scalability of the system network I/O. For HyV we use two VMs per physical machine and give each machine access to one IB port. Figure 11a shows aggregated read throughput (lines/left) and the percentage of operations which take longer than 10μs (bars/right). RAMCloud's performance increases linearly with the number of clients, and HyV shows near native performance. The higher percentage of operations which finished after 10 microseconds with HyV can be explained by general virtualization overhead (e.g. scheduling [13, 20], APIC [8], etc.).

Single Client Latency: Figure 11b shows single client read latency of varying object sizes (30B key) in a single server setup. HyV is able to achieve equal to native latencies. The 99.9 percentile in the virtualized environment is slightly higher due to general virtualization overhead, cf. above.

Summary and Experience: In the above experiments we show that HyV imposes a minimum performance overhead on a distributed RAMCloud deployment by retaining key performance properties of RDMA. Such deployments could directly benefit from virtualization, e.g. by putting standby nodes for recovery in VMs. These nodes can be booted in seconds and moved around at system administrator's discretion. Moreover, by leveraging the VM image provisioning feature, which is only made possible by putting RAMCloud inside a VM, we significantly reduced the system setup and deployment time.

7. Related Work

vRDMA is a paravirtual RDMA device for the OFED RDMA stack [1]. Like HyV, vRDMA allows direct access to application buffers in VMs by RDMA NICs. However, in contrast to HyV, vRDMA uses a paravirtual communication channel to perform data operations. That is, vRDMA cannot completely bypass the hypervisor on data operations. The benefit of such a design is that no device specifics have to be exposed to the VM which may simplify VM checkpointing or VM migration. Additionally, there is no need for memory remapping in vRDMA because it directly uses the OFED kernel Verbs API on the host to forward work requests from the guest to the NIC's work queue. However, it trades these properties for performance (especially latency). From our experience with SoftiWARP we estimate a performance overhead of 3μs on the send path. On the receive path polling for completions cannot be implemented trivially. Either events have to be used on the host or both host and guest have to perform polling which significantly increases the CPU load. In any way, we estimate a total performance overhead of 4μs, in the best case, which e.g. results in up to 4 times higher read latency with IB.

The closest to our solution is "High-performance VMM-bypass I/O in Virtual Machines" [16]. It uses a paravirtual approach to support complete bypass of the hypervisor on data operations. Our work on HyV contributes the following: (1) memory mapping API to allow using unmodified drivers on both guest and host, (2) generic, interconnect independent, RDMA verbs forwarding API to ease development of lightweight virtual providers, (3) demonstration of a real distributed application on a HyV-enabled cluster

Paradice is a I/O paravirtualization framework to virtualize devices that use a (unix-style) device file interface to applications [2]. Like HyV, Paradice virtualizes at a high-level interface ("paravirtualization boundary") to target a larger number of devices in a generic way. Isolation between VMs is achieved by assigning the device to a dedicated driver VM (requires IOMMU) and runtime checks on memory operations. Unfortunately, device drivers have to be modified to perform these checks (tool assisted).

	Direct I/O Path	Hypervisor Awareness	Resource Efficiency	Virtualization Features
Paravirtualization	✗	✓	1^{st}	1^{st}
HyV	✓	✓	2^{nd}	2^{nd}
SR-IOV	✓	✗	3^{rd}	3^{rd}

Table 4: Comparison of virtualization architectures.

8. Discussion and On-Going Work

In the evaluation, we have quantified certain advantages of HyV compared to paravirtualization and direct assignment (e.g., better memory consolidation). In this section, we are discussing both, limitations and potential further advantages of HyV, in the context of traditional virtualization aspects such as resource management, security, isolation or VM migration. Table 4 summarizes some of the findings discussed in this section.

Efficient Resource Management: In comparison to SR-IOV, one key advantage of HyV is its efficient resource management. While it is the explicit resource management of RDMA that allows for efficient resource management in the first place, it is the fact that HyV re-instantiates the hypervisor's role as the global resource manager which extends this feature to virtual machines. RDMA requires explicit resource management where all networking resources such as memory regions, queue pairs, and completion queues etc., are created, managed, and destroyed by the applications running inside a VM. By relaying these resource management calls on the paravirtualized control path, HyV explicitly informs hypervisor of a VM's networking requirements. This is beneficial, considering that hypervisors often struggle to predict a VM's resource usage. For example, as demonstrated, HyV's VM memory footprint can be accounted accurately and efficiently in comparison to SR-IOV/IOMMU's static memory assignment. Similarly, the numbers of QPs – representing connections associated offloaded state on the NIC – can be created and assigned dynamically on-demand to a VM based upon its requirement. Hence, a connection-heavy VM will end up consuming more networking resources on NIC than a connection-light VM. This explicit resource awareness in the hypervisor can also be used for a better VM scheduling and provisioning.

Security and Isolation of VMs: By virtualizing an RDMA NIC, HyV brings the NIC into the security domain of the virtualized infrastructure. HyV depends upon RDMA security mechanisms, which are similar in spirit to what can be found in modern operating systems. All RDMA resources belong to an application allocated protection domain. RDMA hardware enforces isolation by identifying associated protection domains for every resource and restricting access to it, if necessary. Memory access rights (read or write, local or remote etc.) are controlled by the application and enforced by RDMA hardware. The hypervisor can check the validity of these rights at memory registration time. Certain implementation related bugs such as a buggy DMA engine etc., can be contained by using platform virtualization technologies such as IOMMU.

To achieve performance isolation, RDMA provides a number of settings at different granularity depending on the interconnect technology [28]. These facilities can be used to provide QoS, rate limiting etc. among VMs [26] and we are currently exploring similar ideas in the context of HyV.

VM Migration and Checkpointing: VM migration and checkpointing remain challenging issues even for HyV. The complexity arises from the blackbox nature of the offloaded I/O processing in the network card. Clean checkpointing can be achieved by moving a VM into a stable state, where no more offloaded I/O operations are in progress. This state can be achieved by unmapping RDMA resources (or making them read only) inside the VM, and then waiting until all I/O operations and associated completion/error notification events are generated by the RNIC.

VM migration requires explicit support from the RNIC and involves moving network state (QPs, CQs) as well as memory resources (registered memory areas and STAGs). A device-specific hypervisor driver can assist with the state export, migration, and import logic. An alternative approach is to recreate RDMA state and resources on the newly migrated machine [31]. However, implementation complexity of such approaches prohibits their deployment in a distributed, general-purpose virtualized infrastructure.

To our knowledge, neither checkpointing nor VM migration is supported by any RDMA vendor yet. However, we believe that a HyV-like hybrid virtualization architecture retaining control in the hypervisor will be beneficial for implementing these features in the future.

9. Conclusion

RDMA networks – due to their ultra-low latency and high bandwidth – are attractive not only in the HPC domain but also for distributed systems processing cloud workloads in data centers. In this work, we presented HyV, a virtualization framework that unleashes the full performance advantages of RDMA interconnects to virtual machines. HyV implements a hybrid virtualization architecture where network resources are mapped into the guest VM for direct access. We have shown that HyV achieves close-to bare-metal performance inside virtual machines. We further demonstrated the usefulness of HyV in the context of a real distributed system such as RAMCloud, which we deployed on a HyV-enabled cluster equipped with RDMA interconnects.

References

[1] Adit Ranadive and Bhavesh Davda. Toward a Paravirtual vRDMA Device for VMware ESXi Guests. VMware, 2012.

[2] Ardalan Amiri Sani, Kevin Boos, Shaopu Qin, and Lin Zhong. I/O Paravirtualization at the Device File Boundary. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 319–332, New York, NY, USA, 2014. ACM.

[3] Nadav Amit, Dan Tsafrir, and Assaf Schuster. VSwapper: A Memory Swapper for Virtualized Environments. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 349–366, New York, NY, USA, 2014. ACM.

[4] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of USENIX Annual Technical Conference*, pages 41–46, 2005.

[5] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, April 2014. USENIX Association.

[6] Thorsten Von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-net: A user-level network interface for parallel and distributed computing. In *In Fifteenth ACM Symposium on Operating System Principles*, 1995.

[7] Keir Fraser, Steven H, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the Xen virtual machine monitor. In *In 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, 2004.

[8] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. ELI: Bare-metal Performance for I/O Virtualization. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 411–422, New York, NY, USA, 2012. ACM.

[9] InfiniBand Trade Association. InfiniBand Architecture Specification, Volume 1, Release 1.2.1. 2007.

[10] InfiniBand Trade Association. Annex A16: RDMA over Converged Ethernet (RoCE). 2010.

[11] J. Pinkerton J. Hilland, P. Culley and R. Recio. RDMA Protocol Verbs Specification. `http://www.rdmaconsortium.org/home/draft-hilland-iwarp-verbs-v1.0-RDMAC.pdf`, 2003.

[12] Hwanju Kim, Sangwook Kim, Jinkyu Jeong, Joonwon Lee, and Seungryoul Maeng. Demand-based Coordinated Scheduling for SMP VMs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 369–380, New York, NY, USA, 2013. ACM.

[13] Hwanju Kim, Hyeontaek Lim, Jinkyu Jeong, Heeseung Jo, and Joonwon Lee. Task-aware Virtual Machine Scheduling for I/O Performance. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 101–110, New York, NY, USA, 2009. ACM.

[14] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, Ottawa, Ontario, Canada, June 2007.

[15] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.*, 3(2):125–143, March 1977.

[16] Jiuxing Liu, Wei Huang, Bulent Abali, and Dhabaleswar K. Panda. High Performance VMM-bypass I/O in Virtual Machines. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ATEC '06, pages 3–3, Berkeley, CA, USA, 2006. USENIX Association.

[17] Matthew Wilcox. I'll Do It Later: Softirqs, Tasklets, Bottom Halves, Task Queues, Work Queues and Timers. In *Linux.Conf.Au*, 2003.

[18] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-sided RDMA Reads to Build a Fast, CPU-efficient Key-value Store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 103–114, Berkeley, CA, USA, 2013. USENIX Association.

[19] OFED. The Open Fabric Alliance, at `https://www.openfabrics.org/`.

[20] Diego Ongaro, Alan L. Cox, and Scott Rixner. Scheduling I/O in Virtual Machine Monitors. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, pages 1–10, New York, NY, USA, 2008. ACM.

[21] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, New York, NY, USA, 2011. ACM.

[22] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The Case for RAMCloud. *Commun. ACM*, 54(7):121–130, July 2011.

[23] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM. *SIGOPS Oper. Syst. Rev.*, 43(4):92–105, January 2010.

[24] Zhenhao Pan, Yaozu Dong, Yu Chen, Lei Zhang, and Zhijiao Zhang. CompSC: Live Migration with Pass-through Devices. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, pages 109–120, New York, NY, USA, 2012. ACM.

[25] PCI SIG. Single Root I/O Virtualization, at `https://www.pcisig.com/specifications/iov/single_root/`.

[26] A Ranadive, A Gavrilovska, and K. Schwan. FaReS: Fair Resource Scheduling for VMM-Bypass InfiniBand Devices.

In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 418–427, May 2010.

[27] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia. A Remote Direct Memory Access Protocol Specification. RFC 5040, October 2007.

[28] S. A. Reinemo, T. Skeie, T. Sodring, O. Lysne, and O. Trudbakken. An Overview of QoS Capabilities in Infiniband, Advanced Switching Interconnect, and Ethernet. *Comm. Mag.*, 44(7):32–38, September 2006.

[29] Rusty Russell. virtio: Towards a De-facto Standard for Virtual I/O Devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, July 2008.

[30] Animesh Trivedi, Bernard Metzler, and Patrick Stuedi. A case for RDMA in clouds: turning supercomputer networking into commodity. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 17:1–17:5, New York, NY, USA, 2011. ACM.

[31] Vangelis Tasoulas. Prototyping Live Migrationn With SR-IOV Supported InfiniBand HCAs. `http://www.bsc.es/sites/default/files/public/mare_nostrum/2013hpcac-04.pdf`, 2012.