Lightweight and Generic RDMA Engine Para-Virtualization for the KVM Hypervisor

Angelos Mouzakitis, Christian Pinto, Nikolay Nikolaev, Alvise Rigo, Daniel Raho
Virtual Open Systems
17, Rue Lakanal
Grenoble, France

Babis Aronis, Manolis Marazakis
Foundation for Research and Technology – Hellas (FORTH)
Nikolaou Plastira 100
Heraklion, Crete, Greece

***Abstract*—Remote DMA (RDMA) engines are widely used in clusters/data-centres to improve the performance of data transfers between applications running on different nodes of a computing system. RDMAs are today supported by most network architectures and distributed programming models. However, with the massive usage of virtualization most applications will use RDMAs from virtual machines, and the virtualization of such I/O devices poses several challenges. This paper describes a generic para-virtualization framework based on API Remoting, providing at the same time the flexibility of software based virtualization, and the low overhead of hardware-assisted solutions. The solution presented in this paper is targeting the KVM hypervisor, but is not bound to any target network architecture or specific RDMA engine, thanks to the virtualization at the level of the programming API. In addition, two of the major limitations of para-virtualization are addressed: data sharing between host and guest, and interactions between guests and hypervisor. A set of experimental results showed a near to native performance for the final user of the RDMA (i.e., maximum transfer bandwidth), with a higher overhead only to simulate the API functions used to initialize the RDMA device or allocate/deallocate RDMA buffers.**

Keywords - Virtualization, HPC, RDMA, API Remoting

I. INTRODUCTION

Data transfers have always been a main concern in large clusters and data-centres, amplified by a constant request by applications in terms of higher bandwidth and lower communication latency. Remote DMA (RDMA) engines are the response to this type of problem, enabling Network Interface Cards (NIC) to perform DMA-like memory data transfers between nodes of the same computing system. Various network interconnection protocols used in data-centres, such as Infiniband [3] and Ethernet through RDMA over Converged Ethernet (RoCE) [16], are already providing support for RDMA engines. The main advantage of this approach is the drastic reduction of latency, reduced involvement of CPU and thus higher bandwidth compared to other communication paradigms such as network sockets. User libraries based on RDMA transfers are being used for databases, scientific computing and cloud computing in order to optimize communication and data movement between application instances. In parallel, large clusters/data-centres are extensively relying on virtualization as a tool for improved utilization of system resources (e.g., memory, disk, CPU) and hardware consolidation. This is achieved by running multiple virtual instances of a system on the same hardware machine. In addition virtualization is used for resilience thanks to facilities like virtual machines live migration and snapshots.

The virtualization of an I/O peripheral such as an RDMA can be implemented mainly in the following ways: direct device pass-through, exploiting hardware support from the hardware with PCI Single-Root I/O Virtualization (SR-IOV) [10,4] or by para-virtualization [14]. Direct device pass-through, although enabling almost native performance, creates a 1-to-1 mapping between the device and one virtual machine. This means that an RDMA device on a compute node could not be shared among multiple virtual machines, losing the benefits of virtualization in terms of better distribution of available hardware resources. PCI SR-IOV overcomes the problem of sharing the device between multiple virtual machines, but requires support from the hardware that is not always available and reduces the effectiveness of snapshots and live migration. Finally, para-virtualization offers the highest level of flexibility compared with the previous two solutions by being a software-based technique, but suffers from a major drawback: high virtualization overhead due to frequent interactions with the hypervisor and data-sharing handling. RDMA devices can usually be programmed either at the bare metal level, or via a dedicated user-space API. Virtualizing the bare metal interface would lead to a dedicated virtualization solution for each device on the market, while virtualizing at the API level creates a generic solution that can be easily adapted to new APIs, and enables devices using the same programming API to be virtualized with the same implementation of the framework.

In this paper we present a *generic and lightweight RDMA para-virtualization framework for the KVM [5] hypervisor* that overcomes the limitation of hardware assisted solutions, eliminating also the overheads of para-virtualization. The solution virtualizes the RDMA engine at the user-space library level by using a technique called API Remoting, based on an API interception mechanism and a split-driver architecture. The benefits of this solution are threefold:

1. One virtualization framework core for multiple devices/APIs.
2. Native sharing of the device among multiple virtual machines.
3. Low virtualization overhead due to reduced interactions between guests and hypervisor.

From the application perspective the virtualization framework will be completely transparent, since the part of the driver installed in each guest frontend will export the same stub as the original programming API. Internally the frontend intercepts API function calls and re-directs them to the host. On the host side, the second part of the virtualization framework backend is in charge of collecting requests from the various guest frontends to be relayed on the physical device. It then becomes the responsibility of the backend to orchestrate requests from multiple guests, creating the illusion of multiple RDMA devices available on the platform. This approach separates the specific API implementation from the virtualization core, making it easy to extend API Remoting to new APIs. The communication between frontend and backend is ensured by a third component, the transport layer, in charge of actually passing the requests from frontends to the backend process

However, this is not enough for a full solution, since for the virtualization of a device like an RDMA additional factors should be taken into account: interactions with the hypervisor, and guest-host data sharing. The former starts becoming a performance issue when the frequency of interactions is high, and should be minimized since every interaction with hypervisor (hypercall) implies a guest exit that is a renowned expensive operation [18, 7]. The solution presented in this paper reduces the interaction between the virtual machine and hypervisor to the *control-plane* only, while completely avoiding such interactions during regular RDMA operations. The control-plane is implemented in the proposed solution using virtio [15], a well-known para-virtualization framework using circular buffers *vrings* in shared memory for guest-host communication. The second problem, guest-host data sharing also known as the *data-plane*, is also of utmost relevance and in this paper is addressed within the RDMA transport layer. RDMA operations involve data transfer of buffers allocated by user-space applications, and in the bare metal operation of the device do not imply data copies since the RDMA hardware have direct access to those buffers. When virtualization comes into the picture data buffers have to be shared between guest user-space and the RDMA device, and guest-host data copies should be avoided in order minimize the performance loss due to virtualization. In this paper guest-host data sharing is implemented with a zero-copy mechanism [13], enabling true memory sharing between guest and host extended down to the RDMA device.

The RDMA virtualization solution presented in this paper has been tested with an FPGA implementation of an RDMA engine designed for the Unimem Global Address Space (GAS) memory system. Unimem has been developed within the Euroserver [8] FP7 project, to enable a system-wide shared memory abstraction between the nodes of a data-centre. The prototyping system is based on ARMv8 processors, but it should be noted that the API Remoting RDMA virtualization framework has no limitations with respect to the target host processor.

The rest of the paper is organized as follows: Section II provides a comparison with the state-of-the-art solutions for the virtualization of RDMA devices. Section III describes the target RDMA device and its user-space API. Section IV provides the detail of the API Remoting based RDMA virtualization. In Section V a set of experimental results is presented to validate the proposed solution. Finally, Section VI concludes the paper and identifies possible extensions.

II. Related Work

Various studies have been performed on virtualization of RDMA devices, covering both hardware based solutions with the support of PCI SR-IOV [4, 9], and also para-virtualization solutions [2, 12, 14, 11, 7]. Most of the previous work in this field was focused on the InfiniBand architecture [3], followed by RoCE and iWARP.

Musleh et al. [9] demonstrated an RDMA SR-IOV virtualization solution targeting InfiniBand networks, with an overall performance overhead of 15-30% compared to native performance. Also in [4] authors analyzed the SR-IOV virtualization impact on HPC field, with evaluation results for applications based on MPI and Unified Parallel C paradigms.

Authors in [2] have presented a guest driver for the Xen hypervisor to provide DomU support for InfiniBand devices. With hypervisor cooperation, DomU is able to perform direct accesses to device mapped regions e.g., the User Access Region, bypassing hypervisor involvement for queue pairs handling. Data accesses and event handling can be performed directly from the guest user-space application, reducing overhead and featuring performance near to native.

In [14], VMware proposes the vRDMA interface for VMware ESXi guests. Such a para-virtualization framework provides RDMA capabilities to guest applications through a user-space library, compatible with the RDMA ibverbs library [1]. The interfaced library cooperates with the guest device driver, which redirects API calls to the hypervisor. This solution enables also support for virtual machine checkpoints and live-migration (vMotion).

Pfefferle et al. [12] demonstrated the HyV para-virtualization framework for RDMA-capable network interfaces, virtualizing an RDMA device at device driver level. The proposed framework consists of a split driver model, in host and guest kernels, while guest applications can use the OpenFabrics OFED RDMA API [1] to interact with the RDMA device. This hybrid framework supports InfiniBand and iWarp NICs, and the design of the split driver model is able to separate para-virtualized control operations from data operations. Data path operations do not involve the hypervisor,

since the host driver is responsible to provide guests direct access to device resources.

The work presented in this paper has commonalities with the above mentioned studies, but also main differences that go mostly in the direction of a more flexible and hardware agnostic RDMA virtualization solution without sacrificing performance. The main difference with hardware assisted solutions [9, 4] (SR-IOV) lies in the core of the virtualization approach. The solution proposed in this paper is based on software and provides a more flexible virtualization interface, with a better support for device sharing and easily extensible to support virtual machines migration migration. Due to the hardware design of the RDMA device [8] targeted in this paper, the proposed solution has to separate RDMA buffers handling from target network architectures such as InfiniBand, helping the resulting virtualization technology in being agnostic with respect to any target network technology. The RDMA buffers considered in this paper must be contiguous in host physical memory, while in other virtualization works targeting InfiniBand devices [12] RDMA buffers are contiguous in virtual memory. The work presented in this paper overcomes this problem since RDMA buffers are always allocated by the host following device's requirements. Virtual machines will access such buffers through a specific mapping, without the need to know the physical organization of the buffers. Compared to [12] the solution proposed in this paper leverages on the user-space API used to program the RDMA device. With this approach all the devices adopting the same API can be virtualized using the same framework, without the need of major adaptations (e.g., dedicated virtualized kernel device driver). The binary implementation of the API for each specific device is the only requirement behind the approach proposed. The extension to diverse APIs is seamlessly simple, since requires only the fronted library to be adapted together with the corresponding part in the backend. While the core of the virtualization solution can be kept untouched. In addition, even if API Remoting is based on para-virtualization, the global overhead is minimized thanks to the separation of data and control path. With the latter being the only requiring interactions with the hypervisor.

III. RDMA Device Overview

This section provides an overview of the RDMA hardware used for testing and its API exposed to user-space applications. The information in this section is needed in order to better understand the technical choices taken for the design of the virtualization solution.

A. *RDMA device hardware*

The RDMA block is based on the AXI Central Direct Memory Access IP from Xilinx [17]. This block contains a slave AXI4-Lite interface which exports a set of memory mapped registers to configure and program device operations. There are two AXI4 master interfaces: one to perform the transfers between the source and destination memory addresses, and one to fetch DMA descriptors from main memory and initiate memory transfers without CPU involvement. Transfers can be programmed to notify the host CPU in case of success or failure. The RDMA engine performs memory transfers between the nodes of a cluster over a global address space defined by the Unimem memory system.

B. *User-space API*

The RDMA user-space API is centered around objects that represent cluster nodes, DMA buffers and DMA transfers. Each object is associated with a unique identifier. The operations enabled by this API are the following:

- initialization/clean-up of the RDMA library,
- allocation/deallocation of DMA buffers,
- transfer initiation,
- transfer querying,
- association of objects with identifiers, and vice versa.

API calls for DMA buffer allocation and transfer initiation with an asynchronous completion notification are handled with particular attention since these operations have distinct cases in the virtualization context.

DMA memory management calls are responsible to handle the hardware prerequisites, i.e., buffer alignment, and to update the consistent view of DMA buffers among the network. Once a local allocation/deallocation takes place, the connected cluster nodes are informed for the DMA buffer registration/deletion, as a safeguard against incorrect RDMA transfers. The RMDA engine is capable to transfer data between contiguous physical memory regions and, the allocation mechanism must ensure that buffers are physically contiguous. Generic kernel allocators do not guarantee to serve large allocations of contiguous memory (e.g. maximum of 4MiB with the Linux slab allocator). Consequently, DMA buffers (allocation/de-allocation) are handled by a dedicated memory allocator. A global pool for DMA buffers is carved out from the main memory of each of the nodes during the start-up phase, and is later assigned to the DMA allocator. Dealing with physical contiguous buffers creates complications for the virtualization of the RDMA device; however, the solution proposed in this paper overcomes all these problems as described in Section IV-D.

RDMA transfers can be initiated between two allocated DMA buffers, where one of them must reside on the local node. The RDMA initiator can determine the completion status of an active transfer using the transfer polling function, or by registering a callback function. The transfer initiation API calls are always non-blocking and return to the process a unique transfer identifier which can be later used with the polling function. The alternative method is to specify a user-defined function as the transfer callback which will be called upon transfer completion.

IV. IMPLEMENTATION OF THE RDMA VIRTUALIZATION

This section describes the implementation of the proposed virtualization solution, to provide RDMA services to virtual machines running on top of the KVM hypervisor. The virtualization of RDMA implies passing commands from the guest to the host and finally to the device, and also enabling virtual machines to handle DMA buffers by still conforming to the requirements of the RDMA hardware device (e.g. buffers need to be contiguous regions of physical memory).

The virtualization solution demonstrated in this paper is based on *API Remoting*, a software para-virtualization technique enabling sharing of a device among multiple virtual machines, by intercepting API function calls and forwarding them over a cooperating set of software layers from the guest to the host system. Our solution consists of three layers: the backend, the frontend and the transport layer. The *frontend* layer, in the guest, is a shared library that exports the same interface as the original API and forwards user-space applications requests over the transport layer. Passing the RDMA buffer's content between the host and the guest using methods such as TCP/IP sockets would significantly increase the performance overheads, due to memory copies and the complexity of the protocols involved. The proposed *data-plane* is optimized to avoid memory copies which heavily impact the overall performance.

The *backend* layer is a host user process, listening for incoming requests over the transport layer. When it receives a request from a guest, it reads the command and their arguments and programs the device accordingly. Return values and error notifications are propagated back to the source frontend. This layer is the only one actually interacting with the RDMA hardware engine.

The *transport* layer is responsible for delivering data from the backend to the frontend layer, and vice versa. In our case, the transport layer relies on shared memory regions between the host and the guest system to avoid performance degradation due to memory copies.

The overall architecture of the solution is illustrated in Fig. 1, based on the split driver model, with frontend and backend components connected over the transport layer. Guest and host kernels can asynchronously transfer data between them over virtio [15], a para-virtualization framework supporting configurable buffers and asynchronous notifications. The host handles the virtio transport queues (virtqueues) in the kernel through the vhost framework [6]. The host utilizes virtio transfers to inform the guest about a transfer completion event, a synchronization event or to initialize a shared buffer using the guest-to-host zero-copy shared memory framework described in [13]. The guest uses virtio to provide information about shared memory buffers or synchronization events. Transfers over virtio involve a guest exit, which increases the performance overhead. For this reason, only control-plane messages are sent via virtio, while data transfers to and from DMA buffers are handled over shared memory.

A. *Zero-copy Shared Memory Transport Layer*

The transport layer used in this work is based on zero-copy shared memory, to avoid potential performance degradation due to frontend-backend communication and synchronization. The zero-copy shared memory mechanism adopted is based on [13], which has been improved for the purposes of our work.

Shared memory between guest and host is enabled by exploiting a feature of the KVM hypervisor that bases its memory virtualization capabilities on a two-level page table.

In the specific case of an ARM system, a memory access from guest is subject to two memory translations. First the guest virtual address (VA) is translated to a guest or intermediate physical address (IPA), and from IPA to a physical address (PA). An IPA is to be considered as a physical address from the guest's point of view, and as virtual address from the host's point of view. The zero-copy shared memory framework allows buffers allocated in guest memory to be mapped by a user-space process in the host, exploiting the two-level translation. Physical memory pages associated with guest memory are intercepted to be remapped into the host process' virtual memory map. To perform this mapping the host needs to know the guest physical frames (PFN) information for each page composing the buffer to be remapped. Upon request by the host, the guest kernel sends all the PFNs to the host kernel using virtio. Once the host kernel has gathered this information, it can inspect the second-stage page table entries to construct a virtual memory area based on the pages that compose the guest buffer. After the mapping procedure, both processes (in the guest and the host) have in their page tables corresponding entries that point to the same physical memory. It has to be noted that this approach can be used also with other CPU architectures, such as x86.

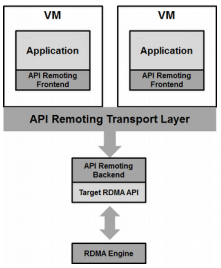

Figure 1. API Remoting abstract infrastructure

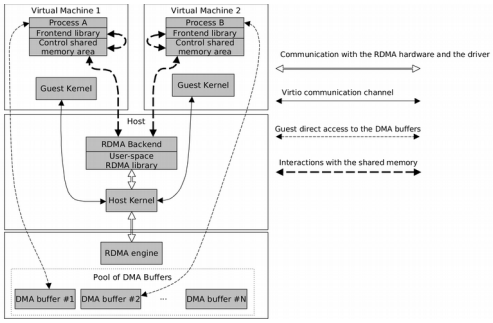

Figure 2. Interactions between the overall architecture components

B. *Frontend and Backend API Forwarding*

This section describes how RDMA API calls are forwarded from frontend to backend, as well as how return values are sent back. At the basis of this implementation there is the shared memory mechanism described in Section IV-A, used for communication between frontend and host backend.

To avoid the performance overhead of existing transport mechanisms e.g., sockets or virtio, the frontend allocates during initialization phase a region of memory that is shared with the backend (see Fig. 2). Such memory region is used by both cooperative processes to exchange information about the forwarded calls without any intervention of the hypervisor.

To forward an API call, the frontend layer stores in the control shared memory area the information related to the specific API function. This information consist of an unique call identifier and the function call arguments. API call identifiers are unique for each function and known to both backend and frontend. Each identifier is an integer that corresponds to an entry in the functions table used by the host backend to perform the correct API call.

The control shared memory area must be protected from concurrent accesses to avoid conflicts. To make sure that accesses will be exclusive, synchronization can take place with memory spin-locks which reside in the shared memory itself. Spin-locks provide a simple solution but could starve the system by taking the whole scheduled process quantum of a process spinning on the lock, thus degrading the overall system performance. An alternative solution is to asynchronously inform the remote process for a synchronization event via virtio. Virtio communication is capable to trigger an interrupt on the remote system. This asynchronous behavior is useful for a synchronization primitive. The caller process (e.g., frontend) will be put into sleep state until the remote process (e.g., the backend) will call for a synchronization event. The procedure of processes synchronization is the following: the local process calls the Forwarding an API call kernel interface to notify the remote kernel for a synchronization event. A virtio message will be transferred to the counterpart driver to indicate a pending synchronization event. When the remote process will call the synchronization event, a virtio message will be send back to the local kernel. In this way only one process will use the shared memory at a time, while the other is put to sleep. However both solutions have drawbacks: spin-locks burn precious CPU cycles, while communication over virtio is slower and involves guest exits. A possible optimization consists of a mixed solution. Synchronization could by default start using spin-locks. In case of long lasting RDMA operations, if no synchronization point is reached after a certain time quantum, the synchronization method can be switched to virtio and the calling process goes into sleep state releasing the CPU. With this approach short RDMA operations can benefit from the high speed reaction provided by spin-locks, while for long lasting operations CPU is not kept active polling on a spin-lock. This will be considered as an enhancement for future extensions of this work.

Sharing of the virtualized device with multiple guest systems is a central point of our para-virtualization solution. The internal structure of the backend process is shown in Fig. 3. A monitor thread listens to a socket interface and is responsible manipulate the guest workers. Each worker thread is associated to one guest identified by the process identifier of the host machine virtualizer (i.e., QEMU). The socket is currently used by frontend processes during initialization to notify the backend of their presence. Upon guest registration a new worker thread is created, and a common initialization procedure takes place: the thread opens a private file descriptor to the extended zero-copy kernel interface. The descriptor is used by the host kernel interface to distinguish the target guest and its resources e.g., virtqueues. A control shared memory area is maintained between each thread and the related frontend process to forward API calls and to place return values.

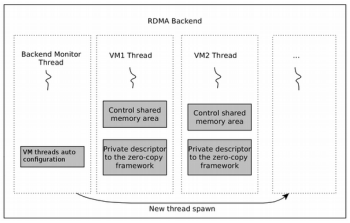

Figure 3. Backend internals

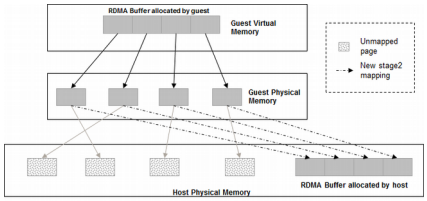

Figure 4. RDMA buffer mapping to guest memory

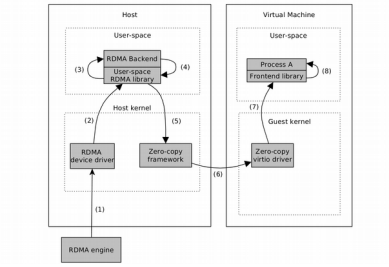

Figure 5. The transfer completion event forwarding procedure.

C. *RDMA Buffer Manipulation*

As described in Section III the RDMA engine targeted by this work is able to transfer data from/to contiguous physical memory regions. However each guest physical address space is allocated from a host user-space application (i.e., QEMU), and from the host point of view is not necessarily physically contiguous. This means that RDMA buffers should be actually allocated by the host, to ensure they are physically contiguous, and then mapped to the guest physical memory space. One possible solution to this memory layout issue is to actually

copy the buffer to be transferred from guest to host, and then program the actual transfer. This solution leads to a performance degradation proportional to the size of the data to be transferred and thus is not a viable solution. In this work this problem is overtaken by extending the existing zero-copy framework in [13], to be able to map buffers allocated from the host into the memory map of a guest user-space process. The goal is to provide guests direct access to RDMA buffers allocated by the host.

Upon the request of allocation of an RDMA buffer the frontend will locally allocate a buffer of the same size, and forwards the allocation call to the host backend together with the PFNs of all the pages composing the buffer just allocated. From the guest perspective the buffer allocated will be seen as the actual RDMA buffer. When the backend process receives the allocation request, it calls the native user-space library to allocate a new RDMA buffer. The allocation is handled by a custom kernel allocator and the memory is physically contiguous and not swappable, thus suitable for RDMA transfers.

Once allocation is done on both sides, all the pages of the actual RDMA buffer allocated in the host are remapped on top of the physical pages backing the buffer allocated by the frontend. To implement this behavior, the guest PFNs received with the allocation call are used to identify the physical pages backing the frontend buffer on the host side, and change the second stage mapping in order to point to the physical pages of the host RDMA buffer (Fig. 4).

The memory mappings of the guest physical memory reside in a virtual memory area (VMA), created by the process that spawned the virtual machine i.e., the QEMU process. The procedure of remapping the RDMA buffer pages to the correct QEMU virtual memory area is handled by a kernel interface, which is an extension of the zero-copy framework in [13]. On the host side, the guest PFNs are converted into host virtual addresses (HVA in Linux terminology). A page walk on the host virtual addresses is sufficient to retrieve the page structures that need to be remapped. The existing pages are unmapped from the QEMU virtual memory, and second stage mapping updated to point to the pages of the actual RDMA buffer. The unmapped pages are also freed so to avoid wasting memory. As result of the allocation procedure, the backend returns to the guest the RDMA buffer object. From now on, guest and host are able to access the same RDMA buffer allocated in the host, and the buffer initially allocated by the guest was used as a place holder to create the page table entries. Note that no memory overhead is added with this procedure, since the pages originally allocated by the guest are freed. Remapping of the memory offers to the guests native access to the DMA buffer, with a slight initial overhead while performing the mapping due to the hardware page-walk of the second stage memory translations.

D. Transfer Completion Event Forwarding

The RDMA device can be programmed to generate a transfer completion event towards the calling process. This behavior has been maintained in the virtualized RDMA framework, and guest applications receive regular transfer completion events. The RDMA library uses a call-back mechanism in order to register a completion event notification. User-space applications register their completion callback that is invoked by the RDMA library when the completion IRQ is signaled by the device. The RDMA driver uses a Unix signal to notify the RDMA library of the event, and in turn call the correct user-registered call-back. In this proposal completion notification is still based on a call-back mechanism, but distributed in two layers. Fig. 5 illustrates how the RDMA completion interrupt is first handled by the RDMA user-space library, and how the backend process forwards it to a guest process. The RDMA interrupt is connected to the interrupt controller of the host CPU, where the host kernel registers a dedicated interrupt service routine. This routine dispatches the IRQ to the RDMA driver handler (1), which informs the RDMA API of the transfer completion using a Unix signal (2). The signal handler is registered during the RDMA library initialization routine, and it is responsible to handle signals from the RDMA s. Once a signal is received, the handler calls the user defined call-back registered by the user-space process when initializing the RDMA transfer (3). In this case the user-space process is the API Remoting backend that registered its own completion backend. The backend process is then in turn able to identify the destination guest (4), and forward the transfer completion event (5). Event forwarding is done by sending to the guest a message over a virtio queue. The message is caught by the guest virtio driver (6) that raises a Unix signal to inform the frontend about the event (7). Finally, the frontend signal handler invokes the completion call-back registered by the guest application (8).

This mechanism enables to preserve the original behavior of the RDMA library, with the minimum possible number of guest exits (only one when sending the virtio message).

V. EXPERIMENTAL RESULTS

This section describes the experiments performed and the results obtained, in order to validate the RDMA virtualization solution proposed in this paper. All the experiments have been performed on a prototype system composed by six nodes equipped with the RDMA device. Each node is based on the ARM Juno R2 development board which the FPGA is loaded with the UNIMEM IP.

A set of test cases has been identified in order to study various aspects of RDMA virtualization based on API Remoting, either at the global level as perceived from the user and also of the individual components of the virtualization framework. The test cases are listed below:

- DMA buffer allocation time
- Maximum bandwidth utilization for variable sized burst transfers
- Overhead of transfer completion event forwarding

- Comparison with the micro-benchmark set of work in [12]

The first three test cases have been executed in two scenarios: 1) *Native*, where the RDMA device is directly accessed from applications running on the host; 2) *Virtualized*, where the RDMA is used by virtual machines via API Remoting. In both scenarios two nodes of the prototype are involved in the data transfers.

DMA buffer allocation: The time needed for an RDMA buffer allocation is different in the native and virtualized scenarios. In the native scenario, the allocation is almost not affected by the size of the buffer to be allocated. In contrast, in the virtualized scenario the time needed to serve an RDMA buffer allocation issued by a guest is proportional to the size of the buffer. The main source of overhead is the page inspection and the remapping procedure, which is necessary to reconstruct the mapping of a physically contiguous DMA buffer in the guest's process address space. To be noted is that this overhead is paid only while allocating the buffer, while all subsequent accesses (read/write) will be served with native performance. Fig. 6 shows the time (ms) needed to allocate an RDMA buffer of size varying from 512KiB to 8MiB.

Bandwidth utilization: Fig. 7 shows the transfer rate obtained in case of single DMA transfer of variable size. While Fig. 8 shows the average bit-rate measured when transferring a total amount of 1GiB, with multiple bursts of variable size. It is immediate to see how in case of a single transfer the performance overhead of the virtualized solution is negligible, reaching its maximum of 7% for a transfer of 512KiB. The transfer time used to compute the transfer rate is measured between the actual guest transfer call, and the moment when the guest application receives the transfer completion notification. Fig. 8 shows the average bandwidth to transfer 1GiB of data in bursts of variable size (from 512KiB to 8MiB). In this case the overall virtualization overhead is proportional with the number of transfers issued. However the highest overhead measured is ~4% with bursts of 2MiB. Even in this scenario, higher completion time is measured with bursts of smaller size. As visible from the graphs, burst transfers have lower virtualization overhead (4% versus 7% in case of single transfer) thanks to the support for multiple outstanding transactions enabled by asynchronous API calls. Applications in the guest receive the transfer ID even if the transfer is not yet submitted by the backend to the device. The backend tracks subsequent transfers from the guest, reducing the overall time required for the submission of burst transfers. This way the virtualization overhead to process one burst is overlapped with the transfer of the previous burst.

Completion notification latency: Asynchronous RDMA transfers register a completion callback that in the case of the virtualized scenario is delayed, crossing the various layers of the virtualization stack. This delay is not fully predictable, since it is affected by the current load of the host and guest system. The completion event forwarding delay has been measured on a system under normal working conditions where there are other processes competing for the CPU. We conclude that events notifications have an average delay of 130 - 140µs. In comparison with an RDMA buffer allocation of 512KiB, the event notification latency is ~240 times lower. This is to highlight that in case of a complete RDMA transfer, the completion notification contribution is negligible over the total transfer time (initialization + data movement + completion notification).

Synchronization: Fig. 9 shows the average time required to synchronize frontend and backend processes using both synchronization techniques, and as expected synchronization with spin-locks is orders of magnitude faster than the primitive based on virtio. This big difference makes the mixed approach described in Section IV-B an interesting enhancement to consider as a next step. In both cases, measurements include the system call overhead used to measure the time elapsed.

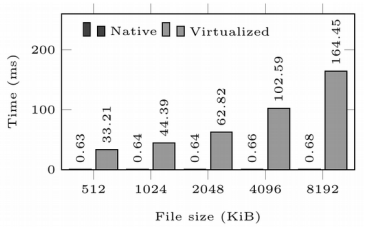

Figure 6. DMA buffer allocation time, from 512KiB to 8MiB.

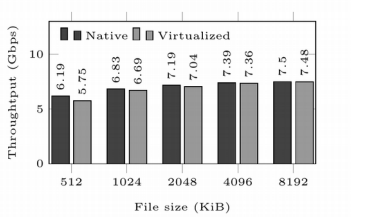

Figure 7. Maximum transfer rate for a single transfer.

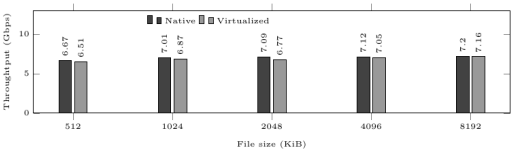

Figure 8. Average bit-rate for a transfer of 1GiB split in burst transfers from 512KiB to 8MiB.

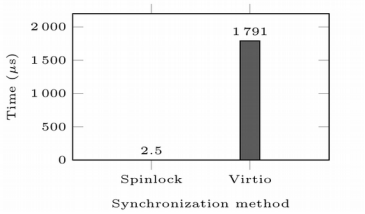

Figure 9. Spinlock and Virtio average time.

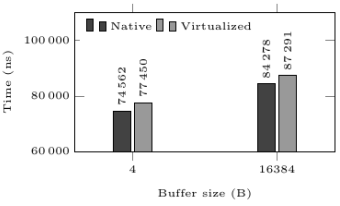

Figure 10. Virtualization overhead on the latency test

In order to quantify the overhead that this RDMA solution introduces with respect to a native RDMA execution, the micro-benchmarks object of this work [12] have been used to measure throughput and latency performance. In [12] the experimental hardware was based on a cluster with Intel Xeon nodes connected through the Mellanox ConnectX-3 VPI 56Gb/s FDR Infiniband RNICs. The HyV virtualization framework showed latency and throughput values similar to a native execution, with message sizes of 4B and 16KB and on the latency test with 1B and 64KB on the throughput. Using HyV, the guest system was able to handle the network's card resources, e.g., the completion poll queues, from the guest process without any hypervisor or OS involvement.

In our case, since the device virtualization is taking place on the user-space API and the framework has no direct access to device's resources, some minimal communication overhead had to be introduced between the function initiator (VM application) and the executor (backend thread) for a control operation. The introduced overhead is near to 3μs and does not depends on to the transfer's size as shown on Fig. 10, 11 using polled transfers. However, the event based solution did not performed as well as the polling counterpart due to the mechanism of software IRQ injection into the guest.

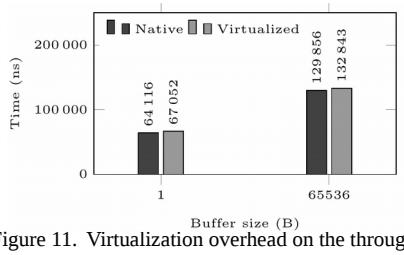

Figure 11. Virtualization overhead on the throughput test

VI. Conclusions

This paper presented a generic RDMA para-virtualization solution based on API Remoting, providing at the same time high bandwidth and low latency transfers to applications running within virtual machines. The separation of the RDMA operation framework between *control-plane* and *data-plane* minimizes the overall overhead to control operations only. In addition the solution presented enables applications running in virtual machines to directly access physically contiguous RDMA buffers, with no virtualization overhead thanks to guest memory remapping. The experimental results discussed in Section V demonstrate close to native performance, and identify memory management as the most expensive type of operations. Further extensions of this work will go in the direction of reducing the latency of completion event forwarding, by injecting a physical interrupt directly to the guest system to reduce the delay caused by virtio communication. Also by monitoring the frequency of the guest's RDMA calls, an adaptive synchronization method will be implemented, with mixed virtio messages and polling spin-locks, in order to reduce the synchronization overhead. Finally, DMA buffer memory management procedures will be studied in-depth to further reduce the virtualization overhead.

Acknowledgment

This work was supported by the *ExaNeSt* project. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 671553. The work presented in this paper reflects only authors' view and the European Commission is not responsible for any use that may be made of the information it contains.

References

[1] *Open Fabrics Alliance. https://www.openfabrics.org*

[2] *Huang, W., Liu, J., Abali, B., Panda, D.K.: Infiniband support in xen virtual machine environment. Tech. rep., Technical Report OSU-CISRC-10/05-TR63 (2005)*

[3] *Infiniband Trade Association: Architecture Specification, Release 1.3. https://cw.infinibandta.org/document/dl/7859*

[4] *Jose, J., Li, M., Lu, X., Kandalla, K.C., Arnold, M.D., Panda, D.K.: SR-IOV sup-port for virtualization on infiniband clusters: Early experience. In: Cluster, Cloudand Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposiumon. pp. 385–392. IEEE (2013)*

[5] *Kivity, A., Kamay, Y., Laor, D., Lublin, U., Liguori, A.: kvm: the Linux virtualmachine monitor. In: Proceedings of the Linux symposium. vol. 1, pp. 225–230(2007)*

[6] *Linux Foundation: Vhost. http://events.linuxfoundation.org/sites/events/files/slides/vhost_sharing_v6.pdf*

[7] *Liu, J., Huang, W., Abali, B., Panda, D.K.: High performance vmm-bypass i/o in virtual machines. In: USENIX Annual Technical Conference, General Track. pp.29–42 (2006)*

[8] *Marazakis, M., Goodacre, J., Fuin, D., Carpenter, P., Thomson, J., Matus, E., Bruno, A., Stenstrom, P., Martin, J., Durand, Y., et al.: EUROSERVER: Share-anything scale-out micro-server design. In: 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 678–683. IEEE (2016)*

[9] *Musleh, M., Pai, V., Walters, J.P., Younge, A., Crago, S.: Bridging the virtual-ization performance gap for hpc using sr-iov for infiniband. In: 2014 IEEE 7th International Conference on Cloud Computing. pp. 627–635. IEEE (2014)*

[10] *PCI SIG: Single Root I/O Virtualization. https://pcisig.com/specifications /iov/*

[11] *Pfefferle, J.: vverbs (2014)*

[12] *Pfefferle, J., Stuedi, P., Trivedi, A., Metzler, B., Koltsidas, I., Gross, T.R.: A hy-brid i/o virtualization framework for rdma-capable network interfaces. In: ACM SIGPLAN Notices. vol. 50. ACM (2015)*

[13] *Pinto, C., Reynal, B., Nikolaev, N., Raho, D.: A zero-copy shared memory frame-work for host-guest data sharing in KVM. In: International IEEE Conference on Scalable Computing and Communications. pp. 603–610. IEEE (2016)*

[14] *Ranadive, A., Davda, B.: Toward a paravirtual vrdma device for vmware esxi guests. VMware Technical Journal, Winter 2012 1(2) (2012)*

[15] *Russell, R.: virtio: towards a de-facto standard for virtual I/O devices. ACM SIGOPS Operating Systems Review 42(5), 95–103 (2008)*

[16] *Subramoni, H., Lai, P., Luo, M., Panda, D.K.: RDMA over EthernetA prelimi-nary study. In: 2009 IEEE International Conference on Cluster Computing and Workshops. pp. 1–9. IEEE (2009)*

[17] *Xilinx: AXI Central Direct Memory Access. https://www.xilinx.com/support/documentation/ip_documentation/axi_cdma/v4_1/pg034-axi-cdma.pdf*

[18] *Zhang, B., Wang, X., Lai, R., Yang, L., Wang, Z., Luo, Y., Li, X.: Evaluating and optimizing I/O virtualization in kernel-based virtual machine (KVM). In: IFIP In-ternational Conference on Network and Parallel Computing. pp. 220–231. Springer(2010)*