Performance Analysis of Ivshmem for High-Performance Computing in Virtual Machines

Pavle Ivanovic¹ and Harald Richter¹
¹Clausthal University of Technology

Email: *pavle.ivanovic@tu-clausthal.de; hri@tu-clausthal.de

Abstract. High-Performance computing (HPC) is rarely accomplished via virtual machines (VMs). In this paper, we present a remake of ivshmem which can change this. Ivshmem was a shared memory (SHM) between virtual machines on the same server, with SHM-access synchronization included, until about 5 years ago when newer versions of Linux and its virtualization library libvirt evolved. We restored that SHM-access synchronization feature because it is indispensable for HPC and made ivshmem runnable with contemporary versions of Linux, libvirt, KVM, QEMU and especially MPICH, which is an implementation of MPI - the standard HPC communication library. Additionally, MPICH was transparently modified by us to get ivshmem included, resulting in a three to ten times performance improvement compared to TCP/IP. Furthermore, we have transparently replaced MPI_PUT, a single-side MPICH communication mechanism, by an own MPI_PUT wrapper. As a result, our ivshmem even surpasses non-virtualized SHM data transfers for block lengths greater than 512 KBytes, showing the benefits of virtualization. All improvements were possible without using SR-IOV.

1. Introduction
High-Performance-Computing (HPC) is traditionally performed on a supercomputer or on a parallel computer. Virtual machines (VMs) or clouds instead are only scarcely used for HPC, because it is the experience of the respective user community that they are not efficient enough for codes based on the Message Passing Interface (MPI) [1], which is the standard for HPC. The reason for that behaviour is that communicating MPI processes are slowed down by the TCP/IP data transmission that is needed for inter-VM and inter-server communication in a virtualized data centre or in a cloud. As a consequence, a parallel computer or supercomputer can typically not be replaced by a local or distributed TCP/IP system. Also, in cases where a single HPC server would be sufficient, a shared-memory communication, which would be much faster than TCP/IP, is not possible because the VMs are isolated from each other. On the other hand, virtual resources and clouds can be obtained in the Teraflops and Terabytes range for a few dozen Euros only, for example from auctions in the Amazon Web Service (AWS), which makes HPC in VMs and clouds very appealing. In this paper, we focus on the performance analysis of communicating MPI processes in two VMs on the same server with MPICH [2] as library. This library is used by us for inter-VM block transfers, via both TCP/IP and ivshmem [3], which is an inter-VM shared memory. The data transfer speeds are compared to those between two non-virtualized MPI processes in host OS, which are using standard SHM shared memory. It could be shown that for larger block sizes the performance is the same for both, virtualized and non-virtualized MPI SHM inter-process communication, thus proving the efficiency of ivshmem. For larger block sizes, ivshmem is even better. The rest of the paper is organized as follows: in chapter
2, an overview on inter-VM communication technologies is given. In chapter 3, our adaption of ivshmem to MPICH is explained from a functional and implementation point of view. Performance results for MPI block transfers between VMs are given and discussed in chapter 4.

2. State-of-the-art in inter-VM communication

There are several technologies publicly available for inter-VM communication inside of the same server, which are: 1.) TCP/IP-based Internet communication, 2.) Virtio-net [4], 3.) vHost-net [5], 4.) virtio-vsocks, 5.) ivshmem and 6.) MVAPICH2-virt [6]. The first five methods are more efficient implementations of TCP/IP-based communication, while the latter two are based on shared memory, which is superior with respect to speed. In this paper, we focus on standard TCP/IP and ivshmem, because TCP/IP is used for inter-VM communication by most MPI implementations, although it provides for the slowest performance. We compare TCP/IP with our ivshmem adaptation, because in contrast to MVAPICH2-virt, it features SHM access synchronization which is indispensable for HPC.

2.1. TCP/IP-based Inter-VM Communication on the Same Server

In standard TCP/IP configuration, a sending MPI process in the guest OS of an emulated PC (source VM) transmits data to a receiving MPI process in another emulated PC (target VM) by calling the send() function of the Berkeley sockets in the source VM TCP/IP protocol-stack. The sender provides the IP address and port number of the receiver as call parameter for send. In case of full hardware virtualization by software, the call results in a guest-OS device driver-call for an emulated Ethernet card, which is intercepted by that QEMU [7] that has launched the source VM. Subsequently, this QEMU hands-over the TCP/IP packet to the single KVM in the host OS kernel which injects the packet into the Internet by means of the host-OS Ethernet device-driver and the real Ethernet card. This happens if no virtual switch (vSwitch) is engaged. Subsequently, the packet returns from the Internet, because the receiver is in the same server. The real Ethernet card accepts its own packet and gives it via its device driver to KVM which forwards it to the target QEMU. Meanwhile the receiving MPI process has called the recv() function in its TCP/IP-protocol stack, which results in a target-VM device-driver-call that is intercepted by KVM. Subsequently, the receiver QEMU hands-over the packet to the target-VM Ethernet device-driver which delivers it to the receiving MPI process. Additionally, source and target QEMUs have to emulate two Ethernet cards (vNICs) as virtual PCI devices for their VMs. The described actions incur a lot of overhead which is bad for HPC. This can be reduced by hardware accelerators for virtualization such as SR-IOV [8], but never fully eliminated. Overhead stems also from the TCP/IP protocol stack which would not be needed, because only a transmission inside of the same server is performed and not a worldwide communication.

2.2. Virtio-net

The KVM hypervisor has two interfaces: the first is used by the QEMUs it cooperates with. The second API is called Virtio that provides for “paravirtualization”. This is a more efficient IO virtualization method than the so-called full software emulation. Besides being an API, Virtio is also a library of paravirtualized device drivers for the guest OS. For example, Virtio-net is the paravirtualized device-driver for a virtual Ethernet card (vNIC). Internet-based inter-VM communication can profit from Virtio-net if a Berkeley send call results in calling Virtio-net. This driver is aware that it is executed in a VM, and it is therefore actively cooperating with its QEMU. With Virtio-net, KVM does not need to intercept guest-OS device-driver accesses to emulated PC devices, because they are not performed. Instead, the parameters for virtio calls are directly forward to QEMU. Technically, the Berkeley send call is not mapped onto a vNIC in guest OS, but via QEMU onto a Virtio-net send queue (“virtqueue”) in host OS. Virtqueues are much simpler and thus faster than vNICs, because they are only buffers. The rest in Virtio-net happens as described in section 2.1. Please note also that the paravirtualized guest-OS device-driver is called front-end driver, while the modified host-OS Ethernet-driver is termed back-end driver. The original host-OS back-end driver cannot be used in Virtio-net because its input is a Linux data-structure called sk_buff, while the front-
end driver outputs so-called virtqueue entries. Both, the front-end and the modified back-end driver, are contained in the Virtio library. The first disadvantage of Virtio is that TCP/IP is still engaged. The second is that the virtqueue entries have to be handled by two QEMUs each, one for the source, the other for the target VM. Finally, for every data frame sent by the real Ethernet card, each QEMU has to make a system call to KVM, which is a time-consuming procedure, because it requires a full process context switch, with all MMU page table entries reloaded.

2.3. vHost-net

VHost-net implements a paravirtualized guest-OS device-driver for an Ethernet card in a more efficient way than Virtio-net does. This is accomplished by outsourcing parts of QEMU's job for managing virtqueues in the user space of host OS to a vHost driver in the host OS kernel. QEMU is still needed for the control plane of the communication, but the data plane, which moves data from the guest OS driver to the real Ethernet card, is completely implemented by the vHost driver. This is possible because the host OS kernel can directly access the virtqueue entries that are output from the paravirtualized guest-OS device-driver, since the VM is a host OS process. This skips QEMU as intermediate stage for every packet. Therefore, Internet-based inter-VM communication can benefit from vHost-net, although TCP/IP is still engaged.

2.4. Virtio-vsock Server

Virtio-vsock is from the user's point of view a simple Linux socket. However, it appears as the same socket in two different operating systems, which is remarkable. One end of this socket is in guest OS, the other in host OS. This is, of course, not directly possible, but requires the support of that QEMU that is responsible for the guest OS. Virtio-vsock thus provides for a communication between guest OS and host OS. Two Virtio-vsocks connected in a row allow for an inter-VM communication with the host OS as intermediate station. The two big advantages of Virtio-vsock are that TCP/IP is not engaged, and that the emulation of two vNICs is also not needed. It allows also inter-VM communication for VMs without vNICs. Virtio-vsock is a newer QEMU feature, but, as every socket, it implies data buffering. Therefore, ivshmem is superior to it with respect to latency.

2.5. Ivshmem

Ivshmem is a virtual PCI device in a guest OS which is emulated by KVM/QEMU. It establishes a Linux/POSIX shared memory (SHM) between the VM and its host OS. Ivshmem enables zero-copy VM-to-Host communication and vice versa, which is very efficient with respect to bandwidth and latency, because no internal data buffer exists. Ivshmem can also be used for inter-VM communication with the host OS SHM as intermediate step. Ivshmem is implemented by mapping its virtual PCI device memory to the host OS SHM. This is possible, because the memory is emulated by QEMU as a data structure inside of itself, and because multiple QEMUs can communicate with each other in host OS, as depicted in figure 1.

2.6. MVAPICH2-virt

Among other features, MVAPICH2-virt implements ivshmem. However, it does not provide for any access-synchronization for it, because, in contrast to our ivshmem remake, it does not make usage of the ivshmem-server. If mutual exclusion in SHM access is needed, then using ivshmem in MVAPICH2-virt means engaging TCP/IP, together with a monitoring protocol in order to emulate that feature. This is still beneficial compared to pure TCP/IP data exchange, but inferior to ivshmem with blocking read synchronization.

3. Our adaption of ivshmem to mpich

Ivshmem was supported for a few years by libvirt and QEMU as a virtual PCI device that allowed for shared memory (SHM) between guests and host. However, this shared memory existed without synchronization for SHM read/write accesses, which was insufficient for most applications. Therefore,
a complementary synchronisation system, called Shared-Memory-Server (SMS) was created later, which became part of QEMU as its ivshmem-server. This server could send interrupts to VMs on the same machine, thus allowing unblocking of an application in a target VM, which was waiting for data from an application in a source VM. Mutual blocking and unblocking was used for SHM access synchronization without consuming CPU cycles (no spinlock), which is indispensable for HPC. However, this valuable feature could only be implemented if both applications were made as the user parts of Linux user-IO device-drivers (uio). Additionally, it was the task of the user to write, by means of code examples of both, the kernel and user part of the uio driver and to integrate his application into this driver. This was a not applicable for HPC with MPI, because MPI processes are never drivers. The reason for ivshmem being limited to uio device drivers was that the one and only possible spinlock-free synchronization between VMs in Linux is possible by means of a blocking read in the user part of uio. As soon as an interrupt arrives at a VM, this user part is called as an interrupt service routine and the blocking read at the target VM is unblocked. Unfortunately, with newer Linux versions, unblocking did not work anymore, with the consequence that ivshmem was again nearly useless because of missing SHM-access synchronization. Although a patch was suggested by the creator of ivshmem, the situation was still difficult, because meanwhile libvirt has also developed further and did not cooperate well any more with ivshmem and the ivshmem-server in QEMU. Finally, no proper documentation existed for ivshmem, which complicated the situation. Our contribution is to patch codes for both, the kernel and the user part of uio device drivers for ivshmem and to find proper version and configuration matches for Linux, QEMU and libvirt so that everything works again. We have also created a documentation and delivered an example usage of ivshmem for HPC by creating a performance test-code based on the integration of ivshmem into MPICH, together with a new SHM communication and synchronization channel inside of MPICH, thereby replacing the inefficient TCP/IP inter-VM communication. In particular, we achieved ivshmem to run for the first time with Ubuntu version 14.04, QEMU version 2.9.50, libvirt version 2.0 and MPICH version 3.2.

![Figure 1. Architecture of ivshmem. The address space of the MPI guest-OS application is first "mmapped" to ivshmem virtual PCI-memory and then mapped to Linux/POSIX SHM.](image-url)

The state that is reached by our work is as follows: we know and documented publicly how to integrate MPICH-based HPC processes to be the user part of uio device drivers: before running the HPC processes, a uio-ivshmem driver patched by us must be loaded into the kernel, together with a proper XML configuration file that supports the so-called doorbell function of ivshmem. Then, it is possible again to use it with blocking read as 5 years ago. In detail: two inter-process communication (IPC) mechanisms are needed inside of the ivshmem-server: 1.) an IPC between the two QEMU instances that are responsible for the source and target VMs. 2.) An IPC between the ivshmem-server and the HPC applications. From the viewpoint of host OS, all codes are user-mode host-OS processes, which simplifies IPC. For the inter-QEMU IPC, Linux eventfds are used. For the IPC between the
ivshmem-server and the HPC applications, sockets are engaged. As soon as a QEMU instance receives an eventfd from another QEMU, it raises an interrupt in its VM with the help of KVM. Additionally, we have engaged Message-Signalled-Interrupts (MSI) in the ivshmem virtual PCI device, thereby providing for the user a numeric value as a user parameter. This requires a proper XML-input configuration-file for libvirt. Finally, in order to integrate ivshmem into MPICH, a custom MPICH was developed that replaces the standard MPI one-sided MPI_Put() call by a wrapper of the same name. If MPI_PUT is called by a HPC code, then our wrapper is executed, which makes all MPICH changes fully transparent to the user and its HPC code. The patched user parts of the uio drivers have two important tasks: 1.) they are mapping ivshmem PCI registers into the address space of the user application, so that the modified MPICH can access them. 2.) They are sending and receiving interrupts to the VMs which in turn call the MPICH application processes. The blocking read () is performed at the target VM on a device file. The PCI-device register-mapping into guest-OS user-space is accomplished by mmap() and ensures a simple SHM access from the HPC code. There are registers and sets in the ivshmem PCI device as follows: 1.) a PCI-BAR0 base-address register which points to a register set, including the so-called doorbell register. 2) A PCI BAR2 register, that points to the SHM. Unblocking a target HPC application process is simply accomplished by writing in our MPICH code an appropriate value to the ivshmem doorbell register, which is accessible by the application and by MPICH.

4. Performance measurements
Performance measurements were accomplished in the same server with varying block sizes and by means of the following three scenarios: a) an inter-VM communication with MPI_PUT for one-sided data exchange and with MPI_WIN_LOCK for passive remote memory access (RMA). This uses the standard MPICH Nemesis-sock channel and results in standard TCP/IP communication. b) An inter-VM communication with MPI_PUT and MPI_WIN_LOCK via ivshmem and SHM synchronization. This required the integration of ivshmem in MPICH. c) A host-OS communication with the standard MPICH via the SHM Nemesis channel. The third scenario was used as a reference for a) and b).

Results for the elapsed times are depicted in figure 2, and for bandwidths in figure 3.

![Figure 2](image2.png)  ![Figure 3](image3.png)

**Figure 2.** Measurements of elapsed time for various message sizes

**Figure 3.** Results of calculated bandwidth for various message sizes

The elapsed time is calculated at the sender by dividing by two the time difference between the first byte sent and the reception of a notification from the receiver about transaction completion, with SHM synchronisation included. Bandwidth is calculated by dividing the elapsed time by the respective message sizes. Based on these measurements, we could observe four phases in figure 2 and figure 3: 1.) Phase 1 for 0 to 4 kB block lengths for the transferred message: in this phase, we can see that the SHM synchronisation time is dominating the data transfer time, because the elapsed time does not increase with an increased message size. Furthermore, the amortization costs for SHM synchronisation are decreasing with increased message size. Finally, ivshmem surpasses TCP/IP by a factor of up to
ten in both, bandwidth and elapsed time, respectively. II.) Phase 2 for 4-64 kB block lengths: in this phase, we can see that the elapsed time increases with the message size, which is visible at the ivshmem curve. The maximum reachable bandwidth is approximately 7.4 GB/s at a block length of 32 KB. Also for TCP/IP, a maximum value for its linear slope is reached. The reason for the latter is that only one TCP/IP packet is needed for all message sizes. Furthermore, the curve for the elapsed time is sharply changing its slope angle. III.) Phase 3 for 64-256 kB block lengths: in this phase, we can see that a transmission saturation is reached at about 256 KB block length with data rates of 2,1 and 0,25 GBs/s. Here, ivshmem surpasses TCP/IP by a factor of four. IV.) Phase 4 for block lengths 512kB: we can see a declining reference curve, while ivshmem becomes the fastest communication. This is a remarkable result, because it means that virtual communication can even be faster than real communication for very large block sizes. The reason for that surprising behaviour is that ivshmem does not make any time-consuming system-call, in contrast to host-OS applications which are coupled by SHM. Ivshmem operates fully in user space because of mmap and the blocking-read feature, which is in the user part and thus in the user space of uio, although the SHM is physically located in host-OS system-space. This is a highly efficient method and because of that, ivshmem is second to none in HPC. As a result, the performance difference between TCP and ivshmem in phase IV is a factor of three with respect to bandwidth, although TCP implementations are highly efficient as well for large message sizes. Finally, it must be mentioned that the block length for the peak ivshmem bandwidth is also influenced by the chosen synchronization and communication type. For example, with active RMA, MPI Start-Post-Wait-Complete and point-to-point MPI_Send, the maximum bandwidth is obtained at a message size of 256kB, resulting in an ivshmem bandwidth benefit four times higher than TCP/IP.

5. Conclusions
We have fixed non-spinlock access synchronization in ivshmem shared memory, which is useful for communication between virtual machines in the same server. Ivshmem is not TCP/IP based, but can replace it, thus delivering a much better performance. Additionally, we made ivshmem runnable with contemporary versions of Linux, QEMU, libvirt and MPICH. Furthermore, an up-to-date documentation for it was provided, including “How-Tos“. Finally, MPICH was adapted by us to our ivshmem remake, and we demonstrated its usefulness by an example HPC code. The MPICH adaptation transparently replaces the MPI one-sided MPI_Put() call, which is frequently used in HPC shared-memory codes. Finally, we have shown by performance measurements, that ivshmem is superior for all message lengths to that MPICH communication type, which is based on a TCP/IP channel. Without our MPICH adaptation, it cannot be avoided that TCP/IP will be used by MPICH for inter-VM communication on the same server, which is useless for HPC. The performance benefit with ivshmem is between factors of 3-10 compared to TCP/IP. For large messages (≥ 512kB), ivshmem even surpasses non-virtualized host-OS Linux/POSIX SHM performance. This means that inter-VM communication can be faster than non-virtualized SHM data transfers for some cases.

References
[1] Gropp W, Lusk E and Thakur R 2014 Using MPI: Portable Parallel Programming with the Message-Passing Interface, Third Edition (MIT Press)
[2] Amer A et al. 2015 MPICH User's Guide, Version 3.2 (Chicago: Argonne National Laboratory)
[3] MacDonell C 2011 Shared-Memory Optimizations for Virtual Machines (Edmonton: University of Alberta)
[4] Tsirkin M, Huck C and Moll P 2013 Virtual I/O Device (VIRTIO) Version 1.0 (OASIS)
[5] Lan S 2016 vHost User Multiqueue Configuration and Use (Intel)
[6] Panda D 2017 MVAPICH2-Virt 2.2 (NOWLAB, Columbus: The Ohio State University)
[7] Weil S 2017 QEMU version 2.10.0 User Documentation (QEMU, www.qemu.org)
[8] Kutch P, Johnson B 2017 SR-IOV for NFV Solutions. Practical Considerations and Thoughts. (Intel, Networking Division)