Efficient shared memory message passing for inter-VM communications

François Diakhaté, Marc Pérache, Raymond Namyst, Hervé Jourdren

Efficient shared memory message passing for inter-VM communications

François Diakhaté[1], Marc Perache[1], Raymond Namyst[2], and Herve Jourdren[1]

[1] CEA DAM Ile de France
[2] University of Bordeaux

Abstract. Thanks to recent advances in virtualization technologies, it is now possible to benefit from the flexibility brought by virtual machines at little cost in terms of CPU performance. However on HPC clusters some overheads remain which prevent widespread usage of virtualization. In this article, we tackle the issue of inter-VM MPI communications when VMs are located on the same physical machine. To achieve this we introduce a virtual device which provides a simple message passing API to the guest OS. This interface can then be used to implement an efficient MPI library for virtual machines. The use of a virtual device makes our solution easily portable across multiple guest operating systems since it only requires a small driver to be written for this device.
We present an implementation based on Linux, the KVM hypervisor and Qemu as its userspace device emulator. Our implementation achieves near native performance in terms of MPI latency and bandwidth.

1 Introduction

Thanks to their excellent isolation and fault tolerance capabilities, virtual machines (VM) have been widely embraced as a way to consolidate network servers and ease their administration. However, virtual machines have not yet been adopted in the context of high performance computing (HPC), mostly because they were incurring a substantial overhead. Recently, advances in hardware and software virtualization support on commodity computers have addressed some of these performance issues. This has led to many studies of how HPC could take advantage of VMs [1].

First, HPC clusters too would greatly benefit from the ease of management brought by virtual machines. Checkpoint/restart and live migration capabilities could provide fault tolerance and load balancing transparently to applications. Moreover, VMs give cluster users a greater control of their software environment without involving system administrators.

Second, it should be possible to use VMs to improve the performance of HPC applications. Using a minimal hypervisor and guest operating system can decrease system noise and therefore greatly increase performance of some communication operations [2]. More generally, virtualization allows the use of specialized guest OSes with scheduling or memory management policies tuned for a specific application class.

Nonetheless, whatever VMs are used for, their integration within HPC environments must have a negligible performance overhead to be successful. Since MPI (*Message Passing Interface*) is the most widely spread communication interface for compute clusters, designing efficient MPI implementations in the context of virtual machines is critical.

Some high performance *Network Interface Cards* can be made accessible directly from within VMs, thus providing near native communication performance when VMs are hosted on different physical machines [3]. However, implementing fast communication between virtual machines over shared memory is also crucial considering the emergence of multicore cluster nodes.

Indeed, to execute MPI applications, it is desirable to use monoprocessor VMs and one MPI task per virtual machine: it allows finer-grained load balancing by VM migration and multiprocessor VMs exhibit additional overheads due to interferences between host and guest schedulers [4,5]. Thus, one major challenge is to implement efficient message passing between processes running inside separate VMs on shared memory architectures.

In this paper, we present a new mechanism to perform efficient message passing between processes running inside separate VMs on shared memory architectures. The main idea is to provide a virtual message passing device that exposes a simple yet powerful interface to the guest MPI library. Our approach enables portability of the guest MPI implementation across multiple virtualization platforms, so that all the complex code dealing with optimizing memory transfers on specific architectures can be reused.

2 Fast MPI communication over shared-memory architectures

In recent years, shared memory processing machines have become increasingly prevalent and complex. On the one hand, the advent of multi-core chips has dramatically increased the number of cores that can be fitted onto a single motherboard. Intel announcement of an 80 core chip prototype shows that this trend is only going to intensify in the future. On the other hand the increase in the number of sockets per motherboard has led to the introduction of NUMA effects which have to be taken into account. As a result, many research efforts have been devoted to achieving efficient data transfers over these new architectures.

In this section, we give an insight on how to perform such data transfers both in a native context, that is between processes running on top of a regular operating system, and in a virtualized environment (i.e. between processes belonging to separate virtual machines).

2.1 Message passing on native operating systems

On a native operating system, data exchanges between processes can be performed in many different ways, depending on the size of messages. Nonetheless, all solutions rely only on two primitive mechanisms: using a two step copy

through pre-allocated shared buffers or using a direct memory-to-memory transfer.

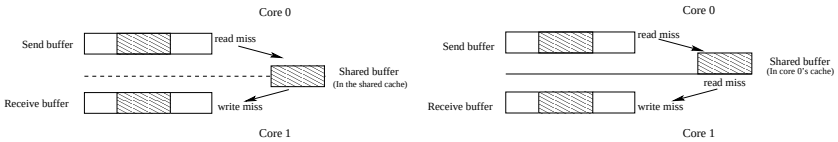
Shared memory buffers: Many MPI implementations set up shared memory buffers between processes at startup thanks to standard mechanisms provided by the OS (e.g. mmap, system V shared memory segments). Communication can then take place entirely in userspace by writing to and reading from these buffers. Several communication protocols have been proposed. In MVAPICH2 [6] all processes have dedicated receive buffers for each peer process: no synchronization is required between concurrent sends to a single process, but n^2 buffers are required and the cost of polling for new messages dramatically increases with the number of processes. To alleviate these issues, Nemesis, one of MPICH2 communication channels [7], uses only one receive buffer per process. This requires less memory and ensures constant time polling with respect to the number of processes. However concurrent senders have to be synchronized, which can be done efficiently using lockless queues.

Buffer based communication makes an efficient usage of the shared caches found in multi-core chips because the buffers can be made small enough so that they fit in the cache. As a result, when communicating between two cores sharing a cache, the extra message copy induces very little overhead (Fig. 1). Moreover, high NUMA locality can be achieved by binding each process to a core and allocating their memory buffers on the closest memory node. Such implementations thus offer very low latencies. However, bandwidth usage is not optimal for large messages, especially when processes are executed on cores which do not share a cache level.

Direct transfer: To avoid the extra copy induced by the use of a shared buffer, some MPI implementations aim at directly copying data between send and receive buffers and thus ensure maximum bandwidth usage for large messages. However, this requires that the copy be performed in a memory context where both send and receive buffers are mapped. That for, a first approach is to perform the copy within the operating system's kernel [8]. This requires a dedicated kernel module which makes this solution less portable. Moreover each communication will incur system call or even page pinning costs that restrict usage of this technique to sufficiently large messages. For smaller messages, shared buffer based communication has to be used. Another approach is to use threads instead of processes to execute MPI tasks [9]. This allows a portable, userspace only implementation but requires that the application code be thread-safe. For example, it may not use global variables.

2.2 Message passing between virtual machines on the same host

The issues raised when trying to provide shared memory message passing between VMs are actually quite similar to those involved in the native case. Indeed, similarly to processes in an OS, VMs do not share the same virtual address space.



(a) Shared cache (b) Separate caches

Fig. 1. Performance impact of using a shared buffer: if communicating cores share their cache (a), the additional copy does not lead to a cache miss and therefore causes little overhead compared to a direct copy. On the other hand, if caches are separate (b), significant memory bandwidth is wasted.

Therefore, one of two things is necessary to allow communications between two VMs:

- Having a pool of shared physical pages mapped in both communicating VMs virtual address spaces, so that the VMs can communicate directly using any shared memory buffer message passing technique seen earlier with no additional overhead. This approach has been studied using Xen page sharing capability to provide fast socket [10] and MPI [3] communication.
- Requesting a more privileged access to the memory of both VMs to perform the message transfers. This avoids unnecessary copies to a shared buffer but requires a costly transition to the hypervisor.

This is once again a tradeoff between latency and bandwidth. To provide optimal performance, the most appropriate technique has to be picked dynamically depending on the size of the message to transmit. Additionally, VM isolation must be taken into consideration. A VM should not be able to read or corrupt another VM's memory through our communication interface, except for the specific communications buffers to which it has been granted access.

3 Designing a virtual message passing device

As we pointed out in section 2, two communication channels have to be provided to allow high speed MPI data transfers between VMs running on the same host: a low latency channel based on shared buffers accessible directly from guests' userspaces and a high bandwidth channel based on direct copies performed by the hypervisor.

To address the challenge of providing these two channels to guest OSes in a portable way, we introduce a virtual communication device which exposes a low-level, MPI-friendly interface.

Indeed, traditional operating systems expect to sit directly on top of hardware and to interact with it through various interfaces. Therefore, they already provide ways to deal with these interfaces and to export them to applications. As a result, to introduce guest OS support for a new device one usually only has to

write a small kernel module or device driver. Moreover, most hypervisors already emulate several devices to provide basic functionality to VMs (network, block device, etc.) Thus, it is possible to emulate a new device without modifying the core of the hypervisor. A virtual device is therefore an easy and portable way to introduce an interface between hypervisors and guest operating systems.

An overview of the usage of these communication channels is provided in Figure 2. We now describe the key features of the device.

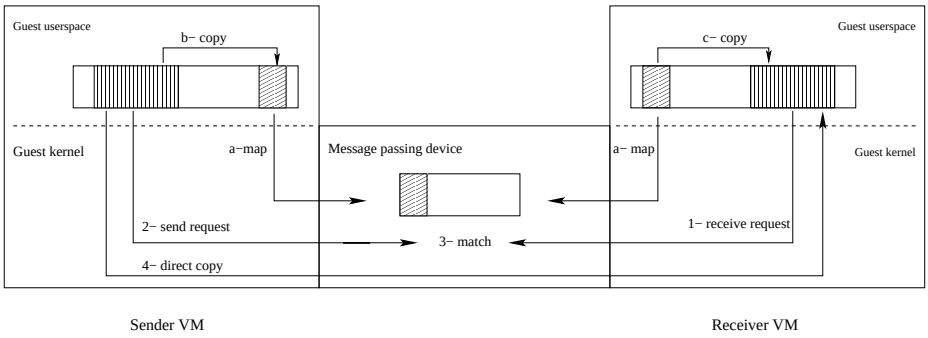


Fig. 2. Overview of the virtual message passing device: shared memory buffers which can be mapped in userspace are provided for low latency communications (a,b,c). Send and receive requests can be posted so that the device performs direct memory copies between VMs (1,2,3,4)

Ports: All communication endpoints on a physical machine are uniquely identified by a port number. This allows several communication channels to be opened on each virtual machine to cater for most use cases. Several MPI processes on a single VM might typically want to use the device for communication with MPI processes running in other VMs. Using port numbers instead of VM identifiers allows to handle these cases seamlessly. A VM must open a port before it is able to issue requests originating from it and only one VM can have a given port opened at a time.

Shared memory: The virtual device possesses onboard memory which corresponds to a shared memory buffer that can be used to communicate between VMs. More specifically, this memory is divided into blocks of equal size and each port is attributed a block. The actual communication protocol is unspecified and shall be implemented in the guest's userspace by the communication library (e.g. MPI library). This way, communications can be performed entirely in userspace and don't incur latency overheads due to context switching.

DMA transfers: The virtual device can process DMA copy operations between arbitrary memory locations on all the VMs that use it. There are 2 types of

requests: *receive* and *send* which both apply to an origin and a target port. They take two additional arguments:

- A list of pointers and sizes which describe a possibly scattered memory buffer to send to or receive from.
- A completion register which is a pointer to an integer in the VM's memory. Completion and failure of a request can be inferred by reading the specified integer. This allows to poll directly from the guest's userspace, thus reducing the number of transitions to the guest kernel and to the hypervisor.

The semantics of these operations are similar to MPI_Irecv and MPI_Isend semantics. In particular, they ensure that a VM can only write to an other VM's memory when and where it is authorized to do so.

Note that this interface is similar to those of high performance network cards which offer buffered and rendez-vous communication channels. This ensures that existing MPI libraries can be ported easily to support this virtual messaging device. While our solution still requires more porting work than a solution offering binary compatibility with the socket interface [10] it is also more efficient, if only because it doesn't incur the system call overhead induced by sockets.

4 Implementation

In this section we provide details on our preliminary implementation of this virtual device using Linux as both guest and host OS and the Kernel Virtual Machine hypervisor. We start by providing a brief overview of KVM and then proceed to describe the implementation of our device from the guest and host point of view.

4.1 The Kernel Virtual Machine

KVM is a Linux kernel module which allows Linux to act as an hypervisor thanks to hardware virtualization capabilities of newer commodity processors. It is thus capable of virtualizing unmodified guest OSes but also supports paravirtualization to optimize performance critical operations. It provides paravirtualized block and network devices and a paravirtualized MMU interface.

Using KVM, VMs are standard Linux processes which can be managed and monitored easily. A userspace component is used to perform various initialization tasks, among which allocating a VM's memory using standard allocation functions and launching it by performing an *ioctl* call on the KVM module (see Figure 3). This component is then asked to emulate the VM's devices: whenever the guest OS tries to access the emulated device, the corresponding instruction traps into the KVM module which forwards it to the userspace component so that the expected result may be emulated. A slightly modified version of QEMU is provided as KVM's default userspace component as it is capable of emulating many devices.

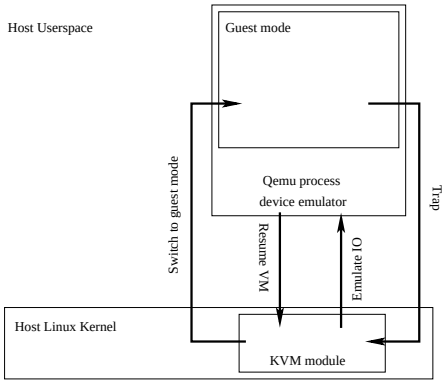


Fig. 3. Architecture of the KVM hypervisor

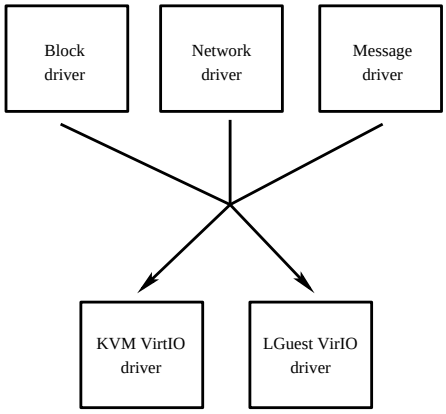


Fig. 4. The VirtIO interface

As a result, it is interesting to note that in this model, implementing the emulation of a new virtual device does not introduce additional code into the host kernel. Everything is performed inside QEMU which is a userspace process.

4.2 Guest implementation

A Linux driver has been implemented to allow Linux guests to handle our device. In the following paragraphs, we describe how this driver accesses the device and how it exposes the virtual device's functions to userspace applications.

Accessing the device: To maximize portability, our device is accessed through the recently introduced VirtIO interface. This interface abstracts the hypervisor specific virtual device handling instructions into an hypervisor agnostic interface. As a result, as shown in Figure 4, the same drivers can be used to handle several hypervisors' implementations of a given device. Only one hypervisor specific driver is needed to implement the interface itself. The VirtIO interface is based on buffer descriptors, which are lists of pointers and sizes packed in an array. Operations include inserting these descriptors into a queue, signalling the hypervisor and checking if a buffer descriptor has been used.

In our virtual device, each open port is associated with a VirtIO queue through which DMA requests can be sent by queuing buffer descriptors. The first pointer/size pair of the buffer descriptor points to a header containing the parameters of the request: whether it is a send or a receive, its buffer size, its destination port and its completion register. The following pointer/size pairs describe the buffer to send or receive.

However there is no VirtIO interface for directly sharing memory between guests. Therefore the shared memory is exposed to guests as the onboard memory of a PCI device.

Userspace interface: Our Linux driver exposes the virtual device as a character device and uses file descriptors to define ports. A port is acquired with the *open* system call, which returns the corresponding file descriptor and released with *close*. *Ioctl* calls are used to issue DMA requests because they require custom arguments such as the destination port and completion register. *Mmap* allows to map the shared memory of the device in userspace.

4.3 Host implementation

The host is in charge of emulating the virtual device. It has to implement memory sharing and DMA copies between VMs. Using KVM, everything can be implemented inside the VMs' corresponding QEMU processes as described below.

Memory sharing between QEMU instances: Since each VM's device emulation is performed by its own QEMU process these processes need to share memory to communicate: DMA requests passed through VirtIO queues must be shared so that send and receives may be matched and each QEMU instance needs to be able to access any VM's memory to perform the copies between send and receive buffers. Moreover, a shared memory buffer must be allocated and mapped as the onboard memory of a PCI device to expose it to guests.

Our QEMU instances are slightly modified so that they allocate all of this memory from a shared memory pool. This is currently achieved by allocating this memory pool in one process with mmap and the MAP_SHARED flag before creating the QEMU instances for the VMs. These instances are then created by forking this initial process.

One limitation of this implementation is that it cannot support a varying number of communicating VMs. However we plan to support this by allocating each QEMU instance's memory separately using a file backed mmap. QEMU instances would then be able to access each other's memories by mapping these files into their respective address spaces. Another possibility would be to run VM instances inside threads of a single process but this would require to make QEMU thread-safe.

DMA transfers: Whenever a guest signals that it has queued DMA requests through the VirtIO interface, the corresponding QEMU instance will dequeue and process these requests. Receives are stored in a per port queue which is shared among QEMU instances and for each send, the corresponding receive queue is searched for a matching receive. If one is found, a copy is performed between the send and receive buffers and the completion registers are updated.

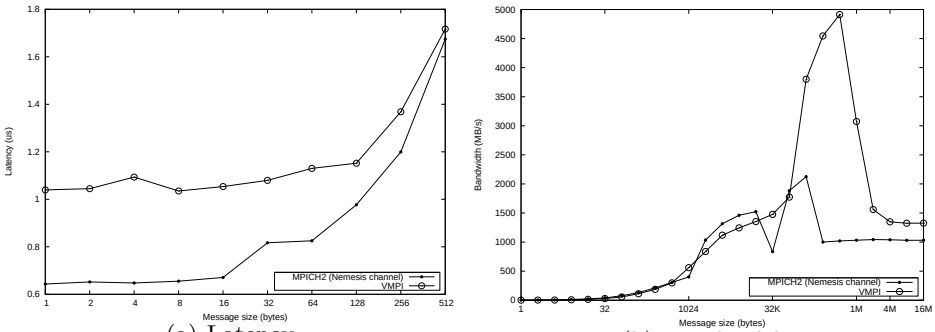
5 Evaluation

To evaluate our virtual device implementation, we have developed a minimal MPI library which provides MPI_Irecv, MPI_Isend and MPI_Wait communication primitives. It should be noted that most other MPI calls can be implemented on top of these basic functions.

Small messages ($\leq$ 32 KB) are transmitted over the shared buffers provided by the virtual device. This 32KB limit has been determined empirically. Each MPI task receives messages in the shared memory block of the virtual port it has been attributed. This memory block is used as a producer/consumer circular buffer and is protected by a lock to prevent concurrent writes.

For larger messages, we use a rendez-vous protocol. The sender sends the message header to the shared buffer of the receiver. In turn, when the receiver finds a matching receive, it posts a DMA receive request to the virtual device. It then sends an acknowledgment to the sender which can post a DMA send request.

We evaluate the performance of our implementation on a pingpong benchmark. We measure the latency and bandwidth achieved between two host processes communicating using MPICH2 and between two processes on two VMs using our virtual device and minimal MPI implementation (VMPI). The machine used for this test has two quad-core Xeon E5345 (2.33Ghz) processors and 4GB of RAM. Processes or VMs are bound to different processors. Results are shown in Figure 5.



(a) Latency (b) Bandwidth

Fig. 5. Results

Nemesis uses lockless data structures and minimizes the number of instructions on the critical path which explains its better performance in small message latency compared to our naive MPI implementation. By implementing Nemesis' communication algorithm in our MPI library we should be able to attain similar performance. The bandwidth graph shows that as long as we use the virtual device's shared buffers for communication (up to 32 KB), there are little difference between the native and virtualized case. For larger messages, the ability to perform direct copies between send and receive buffers allows us to outperform Nemesis even when messages don't fit in the cache. This result outlines an interesting property of virtualization: it can be used to implement optimizations that cannot be performed natively without introducing privileged code.

6 Conclusion and Future Work

In this paper, we presented the design and implementation of a virtual device for efficient message passing between VMs which share the same host. Our evaluation shows that it achieves near native performance in MPI pingpong tests and can outperform native userspace MPI implementations without introducing privileged code on the host.

In the future, we intend to integrate support for our device as a channel in an existing MPI implementation so as to provide the whole MPI interface. This will allow a more thorough evaluation of our solution on real HPC applications. We also plan to extend our virtual device so that it supports additional features such as live migration. This will require a callback mechanism to allow the MPI library to suspend and resume communications appropriately when tasks are migrated. On an other note, we plan to experiment with specialized VM scheduling policies that take communications and NUMA effects into account to reduce overheads when there are more VMs than available cores.

References

1. Mergen, M.F., Uhlig, V., Krieger, O., Xenidis, J.: Virtualization for high-performance computing. SIGOPS Operating Systems Review **40**(2) (2006)
2. Hudson, T., Brightwell, R.: Network performance impact of a lightweight linux for cray xt3 compute nodes. In: Proceedings of the 2006 ACM/IEEE conference on Supercomputing (SC'06). (2006)
3. Huang, W., Koop, M., Gao, Q., Panda, D.K.: Virtual Machine Aware Communication Libraries for High Performance Computing. In: Proceedings of the 2007 ACM/IEEE conference on Supercomputing (SC'07). (2007)
4. Ranadive, A., Kesavan, M., Gavrilovska, A., Schwan, K.: Performance implications of virtualizing multicore cluster machines. In: 2nd Workshop on System-level Virtualization for High Performance Computing (HPCVirt'08). (2008)
5. Uhlig, V., LeVasseur, J., Skoglund, E., Dannowski, U.: Towards scalable multiprocessor virtual machines. In: Proceedings of the 3rd Virtual Machine Research and Technology Symposium (VM'04). (2004)
6. Chai, L., Hartono, A., Panda, D.K.: Designing high performance and scalable mpi intra-node communication support for clusters. In: Proceedings of the 2006 IEEE International Conference on Cluster Computing (Cluster'06). (2006)
7. Buntinas, D., Mercier, G., Gropp, W.: Implementation and Shared-Memory Evaluation of MPICH2 over the Nemesis Communication Subsystem. In: Proceedings of the 13th European PVM/MPI Users Group Meeting (EuroPVM/MPI'06). (2006)
8. Jin, H.W., Panda, D.K.: Limic: Support for high-performance mpi intra-node communication on linux cluster. In: Proceedings of the 2005 International Conference on Parallel Processing (ICPP'05). (2005)
9. Demaine, E.D.: A threads-only mpi implementation for the development of parallel programs. In: Proceedings of the 11th International Symposium on High Performance Computing Systems (HPCS'97). (1997)
10. Kim, K., Kim, C., Jung, S.I., Shin, H.S.: Inter-domain socket communications supporting high performance and full binary compatibility on xen. In: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE'08). (2008)