Xen2MX: Towards High-Performance Communication in the Cloud

Anastassios Nanos and Nectarios Koziris

Computing Systems Laboratory,
National Technical University of Athens,
{ananos,nkoziris}@cslab.ece.ntua.gr

Abstract. Efficient VM communication in Cloud computing infrastructures is an important aspect of HPC application deployment in clusters of VMs. In this paper we present Xen2MX, a high-performance messaging protocol, binary compatible with Myrinet/MX and wire compatible with MXoE. Its design is based on MX and its port over generic Ethernet adapters, Open-MX. Xen2MX combines the zero-copy characteristics of Open-MX with Xen's memory sharing techniques, in order to construct the most efficient data path for high-performance communication, achievable with software techniques. Using Xen2MX, we are able to reduce the round-trip latency to 14μs, compared to directly attached devices (13μs) and to a software bridge setup (44μs).

1 Introduction

Cloud computing infrastructures offer dedicated execution, isolation and flexibility to a vast number of services, providing huge processing power; this feature makes them ideal for mass deployment of compute-intensive applications. However, I/O-intensive applications suffer from poor performance in the cloud context. Numerous approaches have been proposed to alter the programming paradigm of the Cloud [1, 2] leading to a less communication oriented model, though decentralized and distributed.

During the last decade, I/O Virtualization (IOV) techniques have been introduced to provide near-native performance both in 10GbE and exotic interconnection frameworks [3, 4]. Compared to the common case of Virtual Machine (VM) communication (software bridges or routing techniques), IOV has managed to overcome specific limitations in terms of performance. The community has proposed several optimizations to IOV, but still, a major issue correlated with its design remains unsolved: scalability. The number of VMs that enjoy direct data-paths to the network is limited by the hardware capabilities of the specific IOV-enabled adapter. In the era of multi/many–cores, where VM containers would be able to host a great number of VMs, will IOV adapters be able to cope with servicing requests from the system or the network at a sufficient rate ? this problem is multi-parametrical. We believe that close investigation to virtualization system approaches is necessary, using software tools that can mark out efficient data paths and sources of overhead that need to be eliminated.

As we move towards the standardization of Ethernet in both worlds, Cloud computing and High-Performance Computing (HPC), we need a way to to study the effect of message-passing protocols in the Cloud, without having to suffer from TCP/IP's complexity. However, current approaches do not provide a software solution to efficiently exploit hypervisor abstractions to access hardware. In previous studies [5–7] we have attempted to examine the trade-offs related to device sharing using custom lower-level protocols and simple microbenchmarks. We move forward to a more generic design, in order to understand and optimize the way VMs interact with the network in an HPC context.

In this work we describe the design of Xen2MX, a high-performance interconnection protocol for virtualized environments. Xen2MX is binary compatible with MX and wire compatible to MXoE, the Ethernet mode of Myrinet's MX protocol. Although our prototype implementation is in early stages, results from the original MX benchmarks over Xen2MX are promising, reducing the round-trip latency to as low as 14μs compared to 44μs of a software bridge setup, and 13μs of the IOV case. We juxtapose our findings with IOV techniques and examine possible ways to optimize our prototype in order to achieve near-native performance.

2 Motivation and Background

Overview of the Xen Architecture Xen [8] is a popular Virtual Machine Monitor (VMM) that supports Paravirtualization (PV). Data access is handled by privileged guests called *driver domains* that help VMs interact with the hardware based on the *split driver model.* With Single Root I/O Virtualization (SR-IOV) [3], VMs exchange data with the network using a direct VM-to-NIC data path provided by a combination of hardware and software techniques: smart adapters export multiple PCI functions to the VMM; hypervisors assign these functions to VMs, so guest kernels run native drivers and control part of the adapter's capabilities.

In Xen, memory is virtualized in order to provide contiguous regions to OS's running on guest domains. This is achieved by adding a per-domain memory abstraction called *pseudo-physical* memory. So, in Xen, *machine memory* refers to the physical memory of the entire system whereas pseudo-physical memory refers to the physical memory that the OS in any guest domain is aware of.

To efficiently share pages across guest domains, Xen exports a *grant* mechanism. Xen's grants are stored in *grant tables* and provide a generic mechanism for memory sharing between domains. Two guests can initialize an *event channel* between them and exchange events that trigger the execution of the corresponding handlers. This mechanism is often used along with I/O rings, when one guest wants to inform another about the placement of a new request or response in the common ring.

Xen's PV network architecture is based on the split driver model. Guest VMs host the *netfront* driver, which exports a generic Ethernet API to the guest's kernel-space. The driver domain hosts the hardware specific driver and the *netback* driver, which interacts with the frontend using the event channel mechanism and injects frames to the NIC via a software bridge.

Design Choices for a message-passing protocol In setting out to integrate high-performance communication semantics in a virtual environment, we have to respect certain principles: the result of our effort must be portable, simple, scalable, and robust. It must also achieve high-performance in terms of both low-latency communication and line-rate bandwidth. Below, we briefly elaborate on each of these design goals:

Simple and *Portable*: It is rare for system developers to possess the required resources to port their code on various interconnection protocols. It seems that abiding by feature-specific APIs slows down the adoption of new and improved technology. So the obvious choice for them is a widely adopted protocol with a well specified programming interface. The resulting design and code must be simple, understandable and tailored to virtualization demands.

Scalable: It is common case in the HPC world to associate send/receive buffers to any form of communication endpoints (connection oriented or not). This has been widely considered as a bad move [9], especially due to the lack of scalability this implies. As the number of nodes/hosts climb the scale of 1000, then buffers associated with each node are non-negligible in terms of space requirements and management – let alone the mapping of every single host of the network. This seems like a waste of resources compared to just spending a few CPU cycles to poll; it is a fairly significant trade-off we are obliged to take in terms of scalability.

Robust: One of MPI's initial goals was to be kept simple. This is the main reason that MPI's fault tolerance is not elaborate. As a result, the lower-layer interconnection protocol must ensure that error conditions can be handled properly. Connection oriented semantics can solve this issue, keeping the protocol semantics as closely coupled to MPI as possible.

Interconnection Protocol Many custom programming interfaces have been proposed for existing message passing stacks. However, MPI is the current standard for communication on the scientific applications front. For instance, a message passing layer does not need to implement collective communication from scratch – it is only necessary to build the interconnect abstraction of MPI, the Byte Transfer Layer (BTL), that translates MPI semantics into actual calls for the interconnect to handle. One popular lower-level protocol that acts as a BTL for all popular MPI implementations and supports all necessary communication capabilities used by MPI is Myrinet eXpress (MX [10]).

Communication-sensitive applications may obtain significant performance improvement due to dedicated high-speed network technologies. However, as most applications do not yet exhibit communication bottlenecks, many clusters still use commodity networking technologies such as gigabit Ethernet. Indeed, this category of parallel applications is often bound to an implementation of MPI over TCP/IP; nevertheless, TCP/IP has often been criticized as being slow in the context of high-performance computing. As a result, there is a new trend in the HPC market: porting/building high-performance protocols over generic Ethernet.

Open-MX [11] is the software implementation of the MX protocol over generic Ethernet adapters. MX employs user-level networking techniques to achieve high-performance communication. It exploits the capabilities of Myrinet and Myri-10G hardware at the application level while providing low-latency and high bandwidth (less than $2\mu s$ and 1250MB/s data rate). All the actual communication management is implemented in the user-space library and in the firmware. Open-MX follows the same implementation model and semantics as MX; their main difference is that OS-bypass is not possible with generic adapters. Open-MX handles messages the same way as MX, depending on the size of the requested transfer. Its vital building blocks are: *Endpoints*: An endpoint can be considered as a virtualized instance of a device, a logical source or destination of all communications in high-performance interconnects. *Events*: Applications interact with Open-MX through events, a scalable method of communication between kernel-space and user-space. Events may represent receive notifications or send completions. *Regions*: Memory regions are sets of memory *segments* that contain virtually contiguous memory areas allocated by the application. Regions can be the source or the destination of a message and are mainly used in the rendez-vous communication scenario.

When it comes to cloud computing, and specifically virtualization, things get a bit more complicated: virtualized environments are hostile territory for communication-intensive scientific applications. The extra layers of abstraction that comprise the intermediate virtualization layers reduce the overall performance significantly. IOV enabled devices install a direct application-to-hardware data path giving the specific VM the necessary network capacity. Nevertheless, this approach complicates the setup in a way that hardware limitations arise early on the scaling factor and important virtualization features (migration, checkpointing, flexibility in general) are not fully exploited.

3 Xen2MX: Design and Implementation

Our approach proposes the integration of the split driver model described above, to Open-MX. Using endpoints, the asynchronous event mechanism and smart page mappings, VM's user-space is able to communicate with the network using the MX protocol without suffering the overhead of the software Ethernet bridge, while at the same time, the driver domain keeps full control of the network flow. This is actually the core idea of Xen2MX[1]: applications running on VM user-space interact with the MX library, using the MX binary compatibility Open-MX offers; the library makes calls to the frontend module, keeping the protocol semantics intact; the frontend forwards requests to the backend module and vice-versa, depending on the direction of data flow; finally, frames are split and distributed to the frontend's pages or built based on the message size and are transmitted to the network via linux-kernel's Ethernet layers(see Figure 1).

The main communication mechanism between the frontend and the backend is Xen's event channels and the grant mechanism. Memory registration is synchronous, providing the backend with guest user–allocated memory pages. These pages can be attached to socket buffers as fragments, forming a packet to be transmitted to the network. Otherwise, these pages are the destination of a receive request and get populated with data originating from the network.

Endpoints feature send and receive queues, a statically allocated buffer that acts as the source or destination of `MEDIUM` sends and receives (<64KB). These queues, along with their buffers, are mapped in the backend, granted using Xen's mechanisms and addressed by lower-level Open-MX layers.

[1] https://github.com/ananos/xen2mx

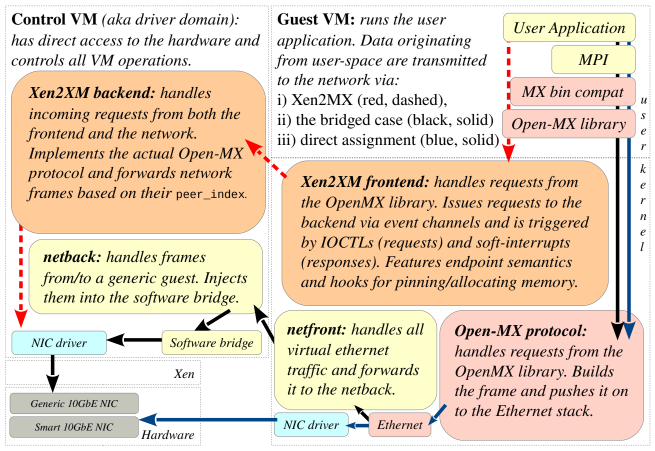
Control VM (aka driver domain): has direct access to the hardware and controls all VM operations.
Guest VM: runs the user application. Data originating from user-space are transmitted to the network via:
i) Xen2MX (red, dashed),
ii) the bridged case (black, solid)
iii) direct assignment (blue, solid)
User Application
MPI
MX bin compat
Open-MX library
user
kernel
Xen2XM backend: handles incoming requests from both the frontend and the network. Implements the actual Open-MX protocol and forwards network frames based on their peer_index.
Xen2XM frontend: handles requests from the OpenMX library. Issues requests to the backend via event channels and is triggered by IOCTLs (requests) and soft-interrupts (responses). Features endpoint semantics and hooks for pinning/allocating memory.
netback: handles frames from/to a generic guest. Injects them into the software bridge.
NIC driver
Software bridge
netfront: handles all virtual ethernet traffic and forwards it to the netback.
Open-MX protocol: handles requests from the OpenMX library. Builds the frame and pushes it on to the Ethernet stack.
Xen
Generic 10GbE NIC
Smart 10GbE NIC
Hardware
NIC driver
Ethernet

Fig. 1. Xen2MX

Frontend–Backend communication: The basic block of our design is how the guest interacts with the driver domain. Xen offers basic communication methods for consumer–producer schemes, on top of which we implement a notification mechanism using both soft interrupts and polling, depending on system demands.

Control data exchange is realized using two cyclic rings, shared between both the backend and the frontend. The first ring handles management commands (requests) and most of the send path. The second ring does completely the opposite: it handles receive notifications. For simplicity, SMALL message data is copied across these rings. This feature, combined with polling on both ends, gives the lowest achievable latency in this setup.

Data exchange for MEDIUM and LARGE messages is realized using a more complicated scenario. As in the Open-MX case, MEDIUM messages use the send and receive queues; their buffers are static, allocated upon endpoint initialization and are addressed using internal indexing. LARGE messages use the registered space (memory regions, see Section 2).

Regions in Open-MX are allocated in user-space and registered using a specific IOCTL. In Xen2MX, the frontend grants the region space to the backend, segment by segment. In the frontend, each segment contains extra fields that keep track of granted pages, which are released only when the backend has finished with them.

The same stands for the send and receive queue buffers. The pages that comprise these queues are granted by the frontend to the backend, resulting in consistent, identical queues addressable by the backend.

The tricky part on this approach is that there is a lot of communication in order to keep regions and queues consistent between the frontend and the backend. The overhead imposed by this communication is non-negligible; therefore, in order to achieve the desirable bandwidth, we need to finetune the way data flow from/to the network. Specifically:

`SMALL` messages are being copied across, so there's not much we can do about it. We keep the number of copies as low as the normal Open-MX, suffering only from the constant overhead of the message being transmitted to the backend.

`MEDIUM` messages are being sent/received via the relevant queues. The situation here is the same as in `SMALL` messages. The added overhead of pinning and granting the send/receive queues is only relevant to the opening/allocation of the endpoint structures which does not occur on the critical path.

`LARGE` messages are send/received via memory regions. However, memory registration is a time-consuming part of the whole process, that needs to be closely analyzed in order to understand the sources of added overhead and overcome them.

On the guest's front, on top of the original allocation and pinning process, we loop around each region segment and grant all pages to the driver domain, keeping track of the relevant grant references. These references (which are actually 32-bit unsigned integers) are packed within a number of reference pages. We then grant the reference pages by messaging the backend. The backend accepts the grants of the reference pages and dereferences their content to obtain the grant references of the original pages. This approach is different that the one used both in Xen's blkback and netback drivers, due to the large numbers of references we export.

Peer/Neighbor discovery: Contemporary message passing protocols, prior to communication, discover the set of network nodes in order to establish the map of the network. When a node comes up in Open-MX, it advertises its existence using *raw endpoints*, a stripped down version of the original endpoint instance. This information is saved in each node's `peer table`, from which any application that wants to communicate with the network gets its `peer index`. This information is essential for upper-layer protocols (such as MPI) which use hostnames to establish initial communication over TCP/IP with their peers.

In our design, peer discovery appears to be more complicated: all VMs that co-exist in the same VM container, share the Ethernet interface(s) of the host machine. Therefore, the network mapping would be incomplete, since multiple peers could not be added to the table using the same interface identification number. To overcome this, we extended the peer table to support multiple peers attached to the same interface.

VM to VM Communication: A common approach adopted by IaaS providers is that the user is not directly aware of the physical placement of the ordered VMs. As a result, the user may end up (by chance) getting his communication-intensive HPC application executed in VMs that co-exist in the same physical machine. If this is the case, then IOV techniques fail dramatically: for a VM to communicate with its peer, data have to flow from the VM's memory to the hardware and then back to the peer VM's memory. This unfortunate situation is not going to happen if we use the split driver model. In Xen2MX, the backend can check the `peer_index` of the destination endpoint, and if it finds it in the local peer table, shared memory communication gets triggered and data get propagated to the peer VM automagically.

4 Preliminary evaluation

In Xen2MX, the user-space library as well as the API are intact. As a result, we use a native MX microbenchmark, `mx_pingpong`, to measure round-trip latency and ping-pong bandwidth between one guest using our implementation and a native linux box, running the latest version of Open-MX[2]. Both machines are identical, featuring one Quad core Intel Xeon 2.4GHz with an Intel 5500 chipset and 4GB of main memory. The network adapters used are two Myricom 10G-PCIE-8A 10GbE NICs in Ethernet mode, connected back-to-back. We deployed our prototype using Xen version 4.2-unstable[3] and linux kernel version `3.4.0`.

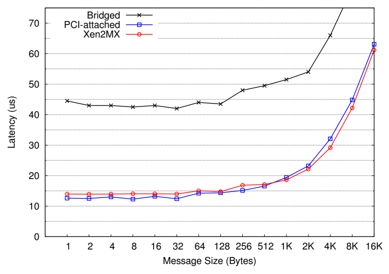

Fig. 2. Latency (lower is better)

We use three different cases to setup our experiment. *Bridged* is the default Xen approach, using a software bridge, *PCI-attached* is the case where the physical device is attached directly to the VM, using Xen's *pass-through* mechanism. *Xen2MX* is our approach.

Figure 2 plots the round-trip latency achieved under the three aforementioned setups. Our approach is almost identical to `PCI-attached` for `SMALL` messages whereas the `Bridged` case achieves 44μs. Using Xen2MX, transferring 1 byte across takes 14μs, less than half of the `Bridged` case and approximately 1μs more than the best measured (`PCI-attached`

[2] Open-MX 1.5.2

[3] changeset 25099:4bd752a4cdf3

case). Figure 3 shows the ping-pong bandwidth measured on the three setups. Our approach outperforms the `Bridged` case which exhibits unstable behavior. After 512KB, performance drops to as low as 10 MB/s, for reasons that we could not explain. We attribute this issue, to the fact that the netback / netfront drivers are not optimized towards high-performance communication; hence this misbehavior when being overwhelmed with buffer handling at this rate.

Our approach seems to follow the scaling of the `PCI-attached` case up until a specific point (32KB). This point is where memory registration becomes noticeable; memory registration is not always on the critical path, depending on the implementation of the protocol. Rendezvous semantics, present in MX and Open-MX message exchange, oblige `mx_pingpong` to register all the memory needed for communication on demand. As a result, the part where the two peers agree on the communication parameters and register memory regions is included in the measurement. We would like to keep the protocol intact, as this is considered the common case in MPI execution, so we are in the process of optimizing the way our initial implementation handles registration and granting.

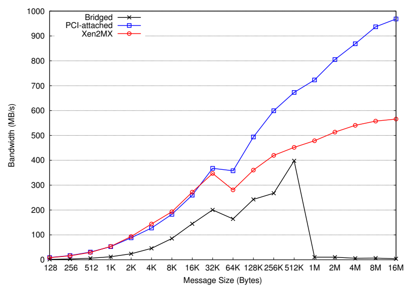

Fig. 3. Ping-pong bandwidth

5 Conclusion and future work

We have presented the design and a prototype implementation of Xen2MX, a software port of the MX protocol to the Xen split driver model, based on Open-MX. Xen2MX benefits from all Open-MX's features to provide binary compatibility with MX as well as wire compatibility with MXoE. Our initial prototype is able to achieve low-latency compared to Ethernet software bridges, the generic case of networking in virtualized environments, as well as comparable results to IOV techniques. Our design is applicable to all paravirtualized approaches and provides an easy, scalable way to deploy communication-intensive applications in the Cloud.

We aspire to perform a detailed analysis of all overheads imposed by virtualization layers in order to optimize our initial implementation. Our future agenda consists of exploiting Xen2MX's binary compatibility with MX (and thus MPI), to deploy HPC applications over this framework and benefit from the flexibility of the split driver design and the scalability of using more than one driver domains. We also plan to closely follow the trends of Ethernet-based message passing protocols such as the Common Communication Interface (CCI [9]) and provide an easy way to adapt our framework to these protocols.

Acknowlegdements This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Heracleitus II. Investing in knowledge society through the European Social Fund. We would also like to thank the anonymous reviewers for their invaluable feedback.

References

1. Murray, D.G., Hand, S.: Scripting the cloud with skywriting. In: Proceedings of the 2nd USENIX conference on Hot topics in cloud computing. HotCloud'10, Berkeley, CA, USA, USENIX Association (2010) 12–12
2. Murray, D.G., Schwarzkopf, M., Smowton, C., Smith, S., Madhavapeddy, A., Hand, S.: CIEL: a universal execution engine for distributed data-flow computing. NSDI'11, Berkeley, CA, USA, USENIX Association (2011) 9–9
3. PCI SIG: SR-IOV (2007) http://www.pcisig.com/specifications/iov/single_root/.
4. Liu, J., Huang, W., Abali, B., Panda, D.K.: High performance VMM-bypass I/O in virtual machines. In: ATEC '06: Proc. of the USENIX '06 Annual Technical Conference, Berkeley, CA, USA, USENIX (2006) 3–3
5. Nanos, A., Koziris, N.: MyriXen: Message Passing in Xen Virtual Machines over Myrinet and Ethernet. In: 4th Workshop on Virtualization in High-Performance Cloud Computing, The Netherlands (2009)
6. Nanos, A., Goumas, G., Koziris, N.: Exploring I/O Virtualization Data paths for MPI Applications in a Cluster of VMs: A Networking Perspective. In: VHPC '10, Naples-Ischia, Italy (2010)
7. Nanos, A., Nikoleris, N., Psomadakis, S., Kozyri, E., Koziris, N.: A Smart HPC Interconnect for Clusters of Virtual Machines. In: VHPC '11, France (2011)
8. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I.A., Warfield, A.: Xen and the Art of Virtualization. In: SOSP '03: Proc. of the 19th ACM symposium on Operating systems principles, NY, USA, ACM (2003) 164–177
9. Atchley, S., Dillow, D., Shipman, G.M., Geoffray, P., Squyres, J.M., Bosilca, G., Minnich, R.: The Common Communication Interface (CCI). In: IEEE 19th Annual Symposium on High Performance Interconnects, HOTI 2011, Santa Clara, CA, USA, August 24-26, 2011. (2011)
10. Myricom: Myrinet eXpress (MX): A High Performance, Low-Level, Message-Passing Interface for Myrinet (2006) http://www.myri.com/scs/MX/doc/mx.pdf.
11. Goglin, B.: High-Performance Message Passing over generic Ethernet Hardware with Open-MX. Parallel Computing **37**(2) (February 2011) 85–100 Open-MX.