Xen2MX: High-performance communication in Virtualized Environments

Anastassios Nanos *Nectarios Koziris*

Abstract

Cloud computing infrastructures provide vast processing power and host a diverse set of computing workloads, ranging from service-oriented deployments to High-Performance Computing (HPC) applications. As HPC applications scale to a large number of VMs, providing near-native network I/O performance to each peer VM is an important challenge. In this paper we present Xen2MX, a paravirtual interconnection framework over generic Ethernet, binary compatible with Myrinet/MX and wire compatible with MXoE. Xen2MX combines the zero-copy characteristics of Open-MX with Xen's memory sharing techniques. Experimental evaluation of our prototype implementation shows that Xen2MX is able to achieve nearly the same raw performance as Open-MX running in a non-virtualized environment. On the latency front, Xen2MX performs as close as 96% to the case where virtualization layers are not present. Regarding throughput, Xen2MX saturates a 10Gbps link, achieving 1159MB/s, compared to 1192MB/s of the non-virtualized case. scales efficiently with the number of VMs, saturating the link for even smaller messages when 40 single-core VMs put pressure on the network adapters.

1 Introduction

Modern cloud data centers provide flexibility, dedicated execution, and isolation to a vast number of service-oriented applications (i.e. high-availability web services, core network services like mail servers, DNS servers etc.). These infrastructures, built on clusters of multicores, offer huge processing power; this feature makes them ideal for mass deployment of compute-intensive applications. In the HPC context, applications often scale to a large number of nodes, leading to the need for a high-performance interconnect to provide low-latency and high-bandwidth communication. Unfortunately, in the cloud context, I/O-intensive applications suffer from poor performance [22, 33, 40], due to various intermediate layers that abstract away the physical characteristics of the underlying hardware and multiplex the application's access to I/O resources. This limitation is one of the most important reasons that HPC applications are not widely deployed in virtualized environments [35].

Numerous studies both in native [8, 16] and virtualized environments [5, 6, 18, 20, 22, 23, 24, 25, 26, 40] explore the implications of alternative data-paths that increase the system's I/O throughput, helping applications overcome significant bottlenecks in data retrieval from storage or network devices. However, near-native I/O performance for Virtual Machines (VMs) in a *generic* cloud environment, built from off-the-shelf components, is still far from being achieved. One of the most important reasons for this shortcoming is the I/O interfaces provided by hypervisors and hardware: software approaches appear too intrusive [4, 17, 19, 20], while hardware approaches need specialized adapters [3, 7, 14, 27].

I/O operations in virtualized environments are handled by software layers within the hypervisor or hardware extensions provided by specialized Network Interface Cards (NICs) (Figure 1). The software layers are implemented either by (a) *emulating* device operations or by (b) the *split driver model*, provided by the *ParaVirtualization (PV)* concept [36]. The hardware approach, case (c), is based on the *device assignment* mechanism, where the hypervisor allows the VM to interact with the hardware directly. Device assignment outperforms the previous methods in terms of high-bandwidth and low-latency [22, 37].

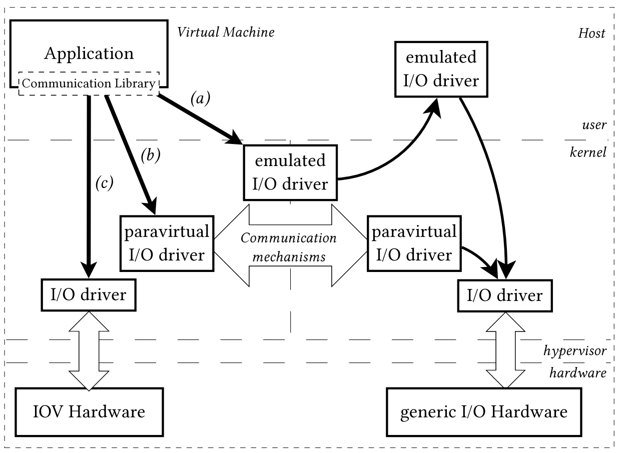


Fig. 1: I/O device access in a typical virtualized environment

Device assignment, however, implies that the adapter is only available to a specific VM, thereby rendering this approach inapplicable to a cloud computing infrastructure, where, by definition, VMs share hardware resources. To address this limitation while providing near-native performance, I/O Virtualization (IOV) techniques have been introduced [3, 19, 26]. These methods combine the advantages of device assignment while allowing multiple VMs to share the same I/O device. The community has proposed several optimizations to IOV, but a major issue related to its design remains unsolved: flexibility. When using IOV adapters, migration becomes more difficult, since the degree of heterogeneity increases between internal or external data-center availability zones. Additionally, the number of VMs that enjoy direct access to the network is limited by the hardware capabilities of the specific IOV-enabled adapter.

This is a serious concern; cloud providers need to be able to manage and manipulate the VM access to the network, in order to account for efficient consolidation, while at the same time providing Quality of Service (QoS) and Service Level Agreements (SLAs). By design, IOV bypasses the VM container, as multiplexing is realized entirely in hardware; although IOV adapters export an interface to control features like traffic shaping and packet filtering, they do not provide a unified way to manage their capabilities. This further complicates the task of managing the network access of a specific range of VMs.

As we move towards the standardization of Ethernet in both Cloud computing and HPC, we need a way to study the effect of message-passing protocols in the Cloud, without the complexity of TCP/IP. However, current approaches do not provide such a software solution to efficiently exploit the hypervisors' interface to the hardware. In previous studies [22, 23, 25], we have attempted to examine the trade-offs related to device sharing, using custom lower-level protocols. In this work, we move forward to a more generic design, in order to gain insight into the system's internals and, ultimately, optimize the way VMs communicate with the network.

We describe the design and implementation of Xen2MX, a high-performance interconnection framework for virtualized environments. Xen2MX contains many features that reduce or eliminate problems associated with traditional PV drivers in an HPC context. Specifically, it minimizes the overhead of event handling for latency sensitive message exchange and enhances throughput by (a) using zero-copy data transfers for large messages (b) re-using existing mappings between guest and host memory regions, (c) decoupling control messages from data exchange using a high-availability consumer-producer scheme. Our design is applicable to any hypervisor that supports PV.

The contribution of this paper can be summarized as follows:

- We identify key design choices for a VM-aware cluster interconnection protocol and discuss current methods of network access in virtualization platforms (Section 2.3).
- We introduce Xen2MX, a high-performance interconnection protocol for virtualized environments, binary compatible with MX and wire compatible with MXoE (Section 3).
- We discover limitations in the existing network approaches present in Xen, leading to increased

communication latency and unstable behavior. Xen2MX is able to overcome these limitations by employing an alternative approach, semantically enriching the guest-to-host communication. Our prototype implementation shows that Xen2MX is able to saturate a 10Gbps link without the necessity to use specialized hardware, at the expense of an $\approx 8\%$ CPU utilization overhead (Section 4).

Specific results from the native MX benchmarks over our framework show that Xen2MX is able to reduce the round-trip latency to as low as 14μs compared to 44μs of a software bridge setup. Compared to directly attached adapters (or IOV), Xen2MX exhibits a 4% overhead. In terms of bandwidth, Xen2MX is able to nearly saturate a 10Gbps link, achieving 1159MB/s, compared to 490MB/s of the bridged case and 1192MB/s of the directly attached case. We also evaluate the scalability of Xen2MX compared to the bridged setup (Section 4.4), while overwhelming the system with send and receive loads from a variable number of VMs (up to 40 VMs): Xen2MX achieves near-native throughput for 512KB messages (16 VMs).

The rest of this paper is organized as follows: first, we lay groundwork in Section 2 by presenting the basic concepts of high-performance computing cluster interconnects. In Section 2.3 we present the requirements for efficient message-passing and elaborate on the current choices for communication in virtualized environments. Section 3 describes Xen2MX, while Section 4 presents a detailed evaluation of its performance, with regards to latency, throughput, CPU utilization, and scaling. Finally, we discuss related research (Section 5) and conclude, presenting possible future endeavors (Sections 6 and 7).

2 Motivation

In this section we describe the components that Xen2MX is based on. We discuss cluster interconnection options for message exchange (Sections 2.1 and 2.2), and focus on the incompatibility between high-performance communication and virtualized environments. We present how our design balances the trade-offs between commodity hardware and near-native performance in a VM context with regards to flexibility.

2.1 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. Apart from basic message exchange, MPI offers collective communication, an important aspect of matrix multiplication, or advanced multi-vector solving techniques – the common case for parallel scientific applications deployed in HPC clusters these days.

A message passing layer does not need to implement collective communication (for instance) 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 nearly all MPI implementations and supports the necessary communication capabilities used by MPI is Myrinet eXpress (MX [21]).

Communication-sensitive applications may obtain significant performance improvement due to dedicated high-speed network technologies. However, there are cases where applications do not yet exhibit communication bottlenecks.

Depending on the user's application domain, communication is an important aspect that imposes significant variability in performance. Some applications can utilize highly parallel systems but do not require a high-performance interconnect or fast storage. One often cited example is digital rendering, in which many non-interacting jobs can be spawned across a large number of nodes with almost perfect scalability. These applications often work well with standard Ethernet and do not require a specialized interconnect for high performance.

On the other hand, there are interconnect-sensitive applications that require low latency and high throughput interconnects. If high-performance networks are not available, many HPC applications run slowly and suffer from poor scalability. Moreover, many I/O-sensitive applications that, without a very fast I/O subsystem, will run slowly because of storage bottlenecks[1].

However, the majority of clusters in the Top500 list, still use commodity networking technologies such as gigabit Ethernet and TCP/IP; these technologies have often been criticized as being slow in the context of HPC. As a result, to overcome limitations in communication operations, without the cost of specialized hardware that suits the HPC market, there is a new trend regarding interconnects: porting/building high-performance protocols over generic Ethernet.

Xen2MX is based on Open-MX [9], a popular Ethernet port pf the Myrinet/MX protocol. In the following section we briefly describe the core concepts of the protocol and elaborate on the available datapaths for communication both in native and virtualized environments.

2.2 Open-MX

MX uses three main message passing methods depending on the message size: messages below 128 bytes (`SMALL`) are written to the adapter using Programmed I/O (PIO); `MEDIUM` messages (below 32KB) are copied to a statically pinned buffer and the hardware transfers them using DMA; messages larger than 32KB (`LARGE`) are transferred using zero-copy via memory pinning, after a rendez-vous process has been established. The sender passes a window handle to the receiver which pulls data from it in 32KB blocks. Multiple blocks may be requested in parallel, and each of them is transferred as a set of eight 4KB-packets by default. 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 2μs and 10Gbps data rate). For this purpose, OS-bypass communication is used along with zero-copy for `LARGE` (>32KB) messages. Therefore, all the actual communication management is implemented in the user-space library and in the firmware. The MX firmware, running on the adapters, is also responsible for matching and directing the data transfer to the correct buffers.

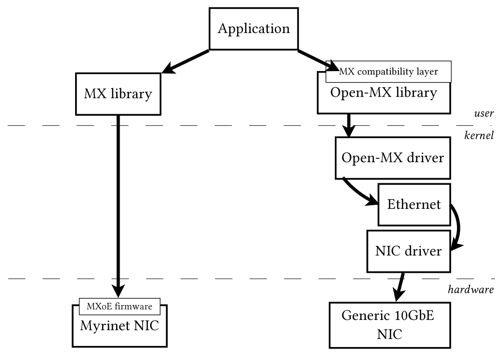


Fig. 2: Design of the native MX and generic Open-MX software stacks.

Open-MX follows the same implementation model and semantics as MX; their main difference is that OS-bypass is not possible with generic adapters. Additionally, the MX wire protocol only specifies the message types, lengths and fragmentation. It does not force Open-MX to use a similar PIO or DMA implementation. Open-MX handles messages the same way as MX, depending on the size of the requested transfer. Open-MX features a userspace library that handles application interaction and binary compatibility with MX as well as a kernel module that handles the protocol processing and the interaction with the hardware. Its vital building blocks are *endpoints*, *events* and memory *regions*.

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. In the Open-MX design, an endpoint contains all the necessary structures that implement the actual communication: send and receive queues, event handles for communication with user-space, memory region information (to track the memory pages that will take part in message exchange) etc.

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.

[1] In a typical cloud environment, a networked storage solution is preferred due to ease of management, elasticity and scalability

Additionally, events trigger timeouts to handle errors relevant to the physical medium (retransmissions) or timeouts to detect a kernel fault.

Regions: Memory regions are sets of memory *segments* that contain virtually contiguous memory areas allocated by the application. These areas consist of non-contiguous physical memory pages which are pinned either on demand or upon creation. Regions can be the source or the destination of a message and are mainly used in the rendez-vous communication scenario.

Open-MX features lower-level abstractions such as region pools; this mechanism implements memory registration caching, a common technique used in many interconnection protocol libraries to provide application buffer reuse and amortize the cost of memory allocation in every send / receive operation. This is essential for Xen2MX as memory registration handling is complicated in a virtualized environment, given the extra layers of abstraction hypervisors provide to the VMs.

The path from the application to the network using a common case Open-MX scenario is summarized in Figure 2. In the following section, we describe the available options for using a protocol such as Open-MX in a virtualized environment, focusing on the extra layers of overhead imposed by virtualized environments.

2.3 Message passing in the Cloud

Virtualization platforms are a hostile environment for communication intensive scientific applications. The extra layers of abstraction that comprise the intermediate virtualization layers reduce the overall performance significantly.

The three basic methods for a VM to interact with a NIC are the following:

(a) device emulation: the hypervisor exports a generic API to the VM using an emulated, generic device;

(b) paravirtual setup: this is an optimized version of the previous approach, which provides the best performance when using generic hardware;

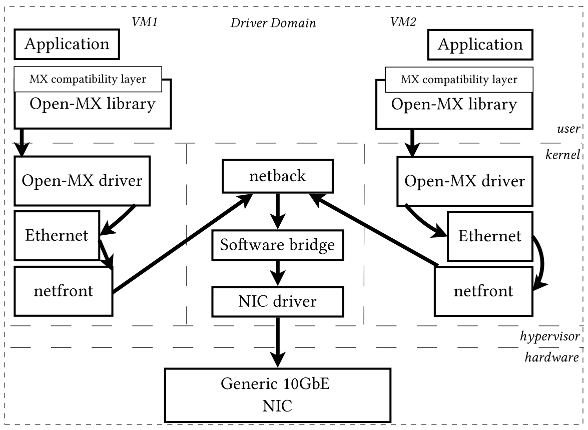


Fig. 3: Paravirtual setup: Open-MX deployed in a common hypervisor, using the split driver model (backend/frontend drivers), a software bridge and generic Ethernet adapters.

(c) IOV: this approach requires specialized adapters that export multiple Peripheral Controller Interconnect (PCI) functions to the host, which, in turn, assigns them directly to the relevant VM.

As seen from the hypervisor's perspective, the data-flow to the network is realized using paravirtual or IOV setups, methods (b) or (c). Emulated devices impose a great deal of overhead on the communication path, whereas specialized devices (IOV enabled) install a direct application–to–hardware data path offering to the specific VM the necessary network capacity.

Figures 3 and 4 show the data path for a common send operation using Open-MX in cases (b) and (c) respectively.

Although IOV, approach (c), outperforms all other cases, it significantly complicates the setup: first, hardware limitations arise early on the scaling factor; the number of virtual network interfaces is limited to the number of PCI functions available on the card. The other obvious problem of this approach is that it requires guests to have hardware-specific drivers. One of the nice features of PV is that it can be used as a hardware abstraction layer; a guest operating system can implement the front-end network

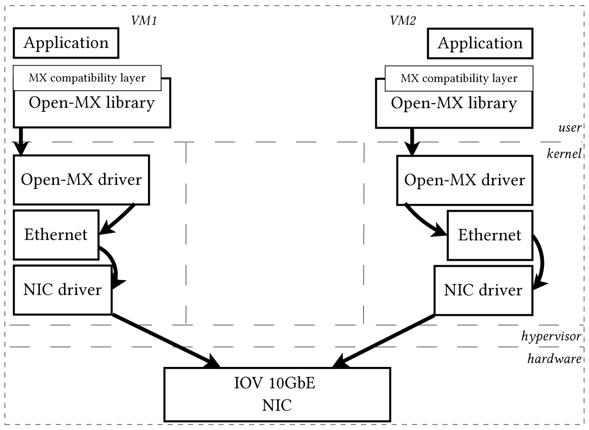


Fig. 4: IOV setup: Open-MX using IOV adapters. Each virtual PCI function is assigned to the relevant VM, installing a direct VM-to-NIC data path, without the intervention of the hypervisor.

interface driver and be able to make use of as wide a range of network hardware as the driver domain supports.

Second, migration becomes more difficult. If a domain is migrated from a machine with a smart NIC to one without, the guests driver needs to be able to move from using the native card to using the Xen front end. This implies that the guest is not able to use a native driver directly; instead, a modified version that wraps the Xen and native interfaces must be implemented. It would be highly inefficient for every interface of this nature to implement this switching code itself, so a standard interface for plugging in multiple interfaces would be useful.

Another problem comes from the fact that communicating between local VMs by sending packets over an external bus is highly inefficient. Additionally, the host is unable to manage VM network access (QoS, access controls, inspection etc.), as multiplexing is realized entirely in hardware. The above imply a significant loss of flexibility. As a result, most current use cases prefer approach (b), the paravirtual setup.

Considering the above arguments, we choose to base our implementation on Open-MX. We design Xen2MX, a networking stack that enables

- VMs user-space to utilize the Open-MX programming interface and exploit all Open-MX features: binary compatibility with MX and wire compatibility with MXoE.
- Cloud providers to benefit from the flexibility that a paravirtualized framework offers compared to IOV approaches which require specialized hardware and increase the level of heterogeneity in the data center.

3 Xen2MX: Design and Implementation

In this section, we present Xen2MX, our framework for high-performance communication in virtualized environments over Ethernet. We structure the communication stack using a frontend running on the guest VM and a backend running on the host. To interface with the application we use a communication library, while to communicate with the network we use the generic linux-kernel Ethernet stack. Our design is based on the Xen split driver model, Myrinet/MX and Open-MX.

3.1 Overview

Xen2MX proposes the integration of high-performance interconnect semantics into existing I/O methods in virtualized environments, namely the split driver model. Using endpoints, the asynchronous event channel mechanism and smart page mappings, applications running on VMs are able to communicate with the network using the MX protocol without suffering the overhead of intermediate virtualization layers, while at the same time, a privileged VM (holding direct access to the hardware), keeps full control of the network flow.

This is actually the core idea of our design: applications running on VM user-space interact with the MX library, using an MX binary compatibility layer; the library issues calls to the Open-MX frontend module, keeping the protocol semantics intact; the frontend forwards requests to the backend and vice-versa, depending on the direction of data flow; finally, frames get split and distributed to the frontend's pages or get built based on the message size

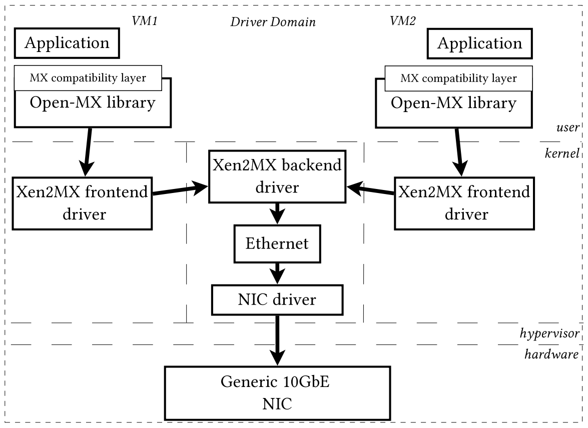


Fig. 5: Xen2MX architecture: we base our design on the split driver model. Xen2MX features frontend and backend drivers, running on VMs and the driver domain respectively. The upper-layer of our framework is fully compatible with Open-MX whereas network communication is realized using generic calls of the linux kernel Ethernet API.

and are transmitted to the network via the Linux kernel's Ethernet layers (see Figure 5).

Xen2MX provides binary compatibility with Myrinet/MX and wire compatibility with MXoE; we keep the protocol semantics intact, and, thus, applications are able to transparently utilize Xen2MX through MPI's MX backend. Messages are categorized as `SMALL` (below 128 bytes), `MEDIUM` (from 129 bytes to 32 KB) and `LARGE` (more than 32 KB).

The most important issue of such a design is the interaction of the different parts of the system in order to provide ultra-low latency while maintaining the same functionality as in native environments. We base Xen2MX internal communication layer on Xen mechanisms, following specific optimizations to alleviate overheads due to memory sharing and synchronous events. Specifically, we optimize the data path by addressing the following issues:

memory handling: Xen2MX uses Xen's grant mechanism to share memory regions from the guest's user-space to the host's kernel-space. These regions contain the portions of memory that are needed for communication. By splitting messages into `SMALL`, `MEDIUM`, and `LARGE` Xen2MX is able to choose between copies and memory sharing in order to effectively exchange messages with the network. Section 3.3 describes in detail how we exploit the grant mechanism to minimize the overhead of the intermediate virtualization layers.

event notifications: A key concept in high-performance interconnection frameworks is the notification mechanism of each peer once a message exchange operation is pending. We combine Open-MX events, along with Xen's event channel mechanism to achieve a tightly coupled producer-consumer scheme between the frontend and the backend (the guest and the host). Section 3.2 explains how the basic blocks of Xen2MX communicate.

data exchange: To efficiently communicate with the network, Xen2MX minimizes the data exchange time. This is achieved by closely examining the data path and eliminating any overheads that arise. These overheads are mainly based on extra memory copies (either in the frontend or the backend); section 3.4 explains how Xen2MX handles message exchange with respect to the optimized data path we build. Section 3.6 describes the whole data path, from the guest's user-space to the network.

3.2 Frontend–Backend communication

The basic block of our design is the interface between the guest and 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; for instance, acknowledgment notifications trigger a lazy handler whereas packet receives (when expected) are delivered immediately.

Figure 6 presents the basic communication semantics of Xen2MX. We implement simple I/O rings based on the grant and the event channels mechanisms. An I/O ring consists of a shared page, granted by the frontend to the backend, and an event channel that provides similar functionality to a virtual IRQ. Each ring is able to carry requests (guest–to–host) and responses (host–to–guest). To send a

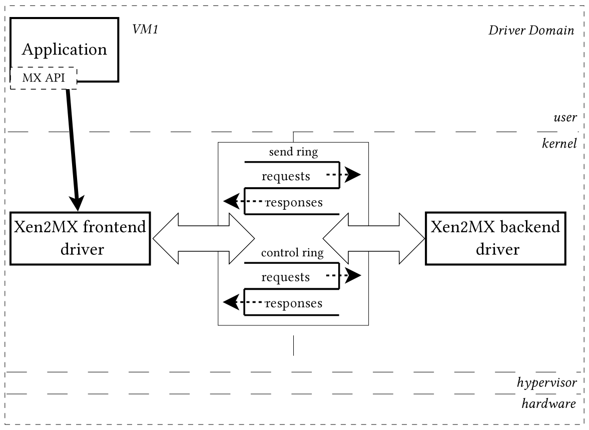


Fig. 6: Xen2MX communication rings: guest–to–host interaction is realized using two ring buffers (control and send I/O rings). Requests originate from the frontend whereas responses are produced in the backend. The rings are implemented using Xen's mechanisms: grants are used for page sharing and event channels for notification (when necessary).

message, the guest builds a request and issues a `PUSH_REQUESTS` command; the guest notifies the host about a pending operation. On the host's front, the relevant handler gets triggered and processes the request. When the host is done processing the request, a response is built and a `PUSH_RESPONSES` command is issued; depending on the direction and the type of the operation the host may notify the guest[2].

Control data exchange is realized using the two rings described above: the `send_ring` handles management commands and most of the send path. The `control_ring` handles receive notifications and acknowledgments.

`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). We further analyze message handling in the following sections.

3.3 Memory regions

Regions in Open-MX are allocated in user-space and registered using a specific `IOCTL`. In Xen2MX, the frontend grants the region space, segment by segment, to the backend. In the frontend, each segment contains extra fields that keep track of granted pages. These pages are released when the backend has finished with them. The way Xen2MX handles regions is shown in Figures 7 and 8.

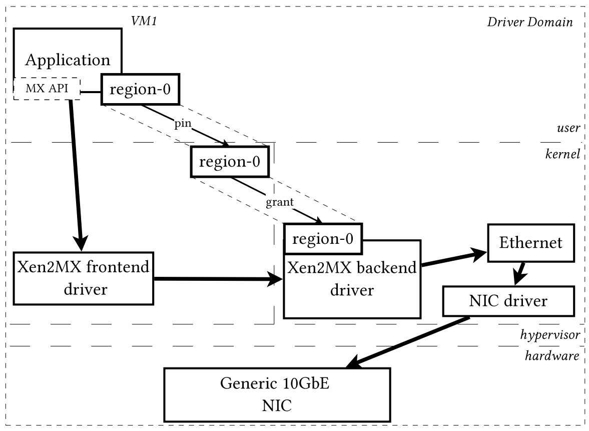


Fig. 7: Regions are allocated in user–space, pinned in the guest's kernel-space and granted to the driver domain. As a result, pages comprising the region are available to the backend directly. The backend can now attach pages to socket buffers or DMA received data into them, avoiding any unnecessary copies that would impose significant performance degradation.

The granting of memory regions to the backend is actually a two step process: on the guest's front, on top of the original allocation and pinning process (done by the Open-MX stack), we loop around each

[2] Receive requests in the backend trigger a notification and are delivered immediately, poking the guest about a pending operation, whereas send acknowledgments are issued and handled by the frontend when the relevant handler is triggered.

region segment and grant all pages to the driver domain, keeping track of the relevant grant references (which are actually 4 byte long positive integers). To gain access to the pages' contents, the backend has to accept these references. Instead of messaging the backend for each one, we pack them into a set of `info_pages`. The frontend grants these `info_pages` using one message and notifies the backend. The backend, in turn, accepts the `info_pages` and dereferences their content to obtain the individual references. Accepting these grants leads to an identical memory region in both ends.

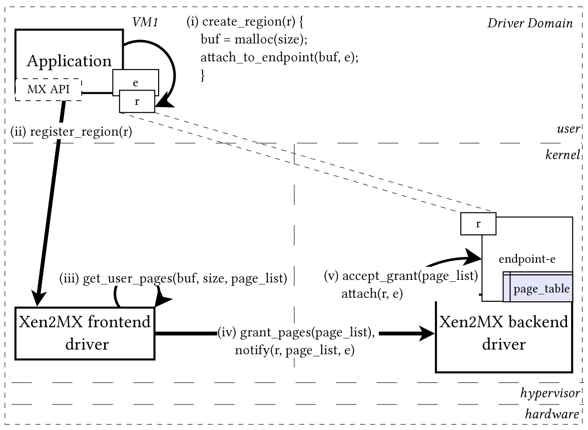


Fig. 8: Basic steps of a region grant operation: (i) the application creates a region, attaches it to the relevant endpoint and (ii) registers it; (iii) the frontend driver pins the pages that comprise each segment of the region and (iv) grants them to the backend; (v) the backend driver accepts the grant, and updates the relevant structures. From this time on, the VM's data are available on the backend.

To clarify the above procedure we consider the following example. To grant an 8MB segment we need $\frac{8MB}{PAGE\_SIZE}$ references. For a 4K `PAGE_SIZE`, we need 2048 references, leading to $\frac{2048*4bytes}{4096} = 2$ `info_pages`.

1. The frontend allocates the `info_pages` and loops around the 8MB segment to grant each of the 2048 pages to the backend, populating the `info_pages` with the relevant `grefs`. The `info_pages` now contain the grant references that need to be transferred to the backend.
2. The frontend grants the `info_pages` to the backend and notifies it for a pending operation.
3. The backend accepts the grant and digs in the `info_pages` for the actual `grefs` of the region segment.
4. After getting the references, the backend accepts the grants, and, thus, the 8MB segment is available in both ends of the split driver.

This approach is different than the one used both in Xen's blkback and netback drivers, due to the large numbers of references we need to export. This is essentially a semantic difference: the netback/netfront drivers handle operations in an MTU granularity (since every operation involves an Ethernet frame transmit or receive). Our approach handles operations based on an interconnect concept since we design it for high-performance communication.

3.4 Message exchange

A virtualized environment lacks the flexibility of following generic OS rules to optimize data exchange with the network. For instance, *zero-copy* in a virtualized environment is implemented in a completely different way than the one in a typical OS. The extra step involved is moving / sharing data between the guest and host. We carefully design the stack, in order to keep MX compatibility *and* eliminate intermediate overheads.

`SMALL` messages are copied from user–space. Since their size is smaller than 128 bytes, granting a page would impose a significant overhead, as this operation involves at least two hypercalls as well as the messaging overhead (frontend to backend). We choose to pass data along with the original request via the `control ring`; however, instead of copying one more time, we transfer data from user–space directly to the request buffer. Thus, we are able to keep the number of copies as low as 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 mechanism used for these messages is the same as in SMALL messages. The added overhead of pinning and granting the queue buffers is only associated with the opening / allocation of the endpoint structures, which does not occur on the critical path. Data is placed into the queue from user–space, passing only the index and the message size via an IOCTL. The MEDIUM message request is then pushed to the backend, so data follow the common Open-MX path, with the extra overhead of the guest–to–host notification.

LARGE messages are sent/received via memory regions. In order to use a region, one has to register it. This is the tricky part of the process: memory registration is a time consuming operation, due to the large number of pages being granted by the guest. Moreover, to establish the actual mapping, the host has to accept the grant. Detailed explanation of this procedure and the overhead these operations impose is given in Section 4.2.1.

3.5 Send and receive queues

Open-MX uses statically pre-allocated buffers to provide MEDIUM message communication. These buffers are allocated upon endpoint creation and are indexed using internal endpoint fields. However, in the Xen2MX case these buffers must be accessible both from the backend and the frontend. To enable this functionality, the frontend driver *grants* the pages that comprise these queues to the backend, resulting in consistent, identical queues addressable by both drivers. To account for queue index integrity, endpoints in the frontend are also mapped to the backend on demand.As a result, the backend is able to place data originating from receive requests to the receive queue and notify the frontend; at the opposite direction, the backend sends data to the network from the send queue immediately after the frontend has populated the relevant memory space.

3.6 User–space to network

As mentioned earlier, we have designed Xen2MX to keep the virtualization overheads to a minimum, by closely examining every aspect of communication (user–to–kernel, guest–to–host and finally host–to-network). Figure 9 demonstrates the frontend's capabilities in terms of communication with both the user–space application and the backend.

Specifically:

- *user–to–frontend*: The application issues standard Open-MX API calls via IOCTLs. The frontend calls the necessary handler and, via a state machine, processes the request accordingly. For instance, if the request is ENDPOINT_OPEN, the frontend allocates the necessary structures and grants all relevant pages to the backend.
- *guest–to–host*: The frontend places a request into the relevant ring (which is actually a shared page) and notifies the backend that a pending request is due (PUSH_REQUESTS) via Xen's event channel mechanism. The backend gets notified and invokes the handler to process the request. When the backend is done, it pokes the frontend to notify for completion. In the example above, the backend opens an endpoint, attaches it to the relevant interface, accepts the page grants and builds an acknowledgment response pushing it back to the frontend.
- *host–to–network*: The backend issues Open-MX calls to send data to the network using the generic workflow: packet processing and submission to the linux Ethernet layer.

3.7 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. The node broadcasts its hostname, followed by an identification number of the physical interface(s) attached. 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

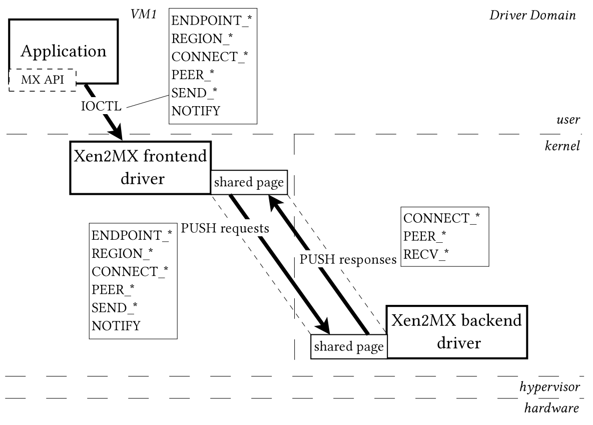


Fig. 9: Driver capabilities that form the user–to–network data path.

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 extend the peer table to support multiple peers attached to the same interface. Upon initialization, the backend maps all VMs to its attached interfaces, while building the rest of the network map. Thus, remote Open-MX packets flow to a VM based on the MAC address of the host machine, while local Open-MX packets (originating from other VMs in the same VM container) are forwarded to the relevant VM.

3.8 VM to VM Communication

A common approach adopted by IaaS providers is that the user is not aware of the physical placement of the ordered VMs; moreover, the user has probably no choice on the VM placement either. As a result, the user may end up with VMs that co-exist in the same physical machine, executing communication intensive workloads. If this is the case, IOV techniques fail dramatically: for a VM to communicate with its peer (that is physically co-located in the same VM container) data has to flow from the VM's memory to the hardware and then back to the peer VM's memory. Unfortunately, since multiplexing is done entirely in hardware, there is no easy way for the host to identify such situations and switch to local communication paths.

This is not the case when using a paravirtual approach, such as Xen2MX. Our design is based on the fact that application endpoints co-exist in the backend (the helper module of the driver domain). This way, the backend can look up the `peer_index` of the destination endpoint, and trigger the shared memory communication mechanism. Data then get propagated to the peer VM automagically. To account for optimizations on the shared memory mechanism we identify two options for data exchange:

Copy: data can be copied across to the peer VM, based on the regular data flow technique we use for standard send or receive.

Flip: data that originate from frontend pages physically exist in machine pages (identified by their frame number, `mfn`). Considering a case where these `mfn`s can populate the peer VM's pages, data from the sender VM can reach the peer VM without a single copy.

We choose to keep the copy approach for `SMALL` messages (< 32KB) and page flipping for `LARGE` messages.

4 Performance Evaluation

In this section we describe the experiments we performed to analyze the behavior of Xen2MX and identify specific characteristics of our approach compared to the common communication setups used in virtualized environments.

We setup two identical boxes described in Table 1 and perform two basic experiments, in order to illustrate the merits and shortcomings of our approach compared to the other methods of communication, as well as the performance in a real-life scenario:

- **Experiment 1** *VM container to native Linux*: Using this setting we closely evaluate our prototype implementation while overwhelming the system with data (send and receive) from a native, generic Linux box. This way, we are able to quantify in great precision the size of the overhead our solution incurs, compared to the non-virtualized case. We setup one VM container and spawn up to 8 single core VMs (VM_1...VM_8) on $Host_0$. We leave 4 cores for the driver domain. The second host ($Host_1$) is a generic Linux installation (no virtualization involved). We use `mx_pingpong`, a native MX micro-benchmark that exchanges messages of variable lengths between MX endpoints and deploy it between $Host_1$ (8 separate instances) and VM_1 through VM_8 (one instance per VM).

- **Experiment 2** *VM container to VM container*: This setting illustrates the system performance in a real-life scenario. We setup both VM containers as above: (VM_1...VM_8 on $Host_0$ and VM_9...VM_{16} on $Host_1$). We then deploy `mx_pingpong` between opposite VMs (VM_1 to VM_9, VM_2 to VM_{10} etc. – one instance per VM).

4.1 Basic setup/testbed

The API, as well as the user-space library, which handles most of the protocol issues, are intact; this is a design choice, in order to keep our protocol MX binary compatible. As a result, we are able to use native MX microbenchmarks to evaluate Xen2MX and compare its performance to the respective non-virtualized and generic setups.

We use three different cases to setup our experiments. `Bridged` is the default Xen approach, using a software bridge to transmit/receive frames and paravirtualized drivers to export virtual Ethernet interfaces to the VM. `PCI-attached` is the case where the physical device is directly attached to the VM, using Xen's *pass-through* mechanism. This approach leads to performance as close to native as possible, since there is no multiplexing involved, either by the hypervisor, or the firmware of the NIC itself. In most cases, this approach performs better than the IOV case. Unfortunately, at the time of the experiment, we were unable to use a Single Root I/O Virtualization (SR-IOV) adapter but, in terms of performance numbers, this case provides an upper bound for IOV. `Xen2MX` is our framework; we deploy two settings: `Xen2MX-plain` is the generic approach where we disabled memory registration caching; `Xen2MX-tuned` is the optimized one where we enable memory registration caching (`MX_RCACHE=1`). We perform extensive tests using both settings (memory registration cache on and off) for the other two setups (`Bridged` and `PCI-attached`) and found no significant variability on the results (less than 1% difference); thus, in what follows, we plot the best of each case. For completeness, we plot curves for the `Native` case (no virtualization involved), in order to point out parts of overheads that are not due to our framework.

	VM containers/Native Linux boxes
Processor	2x Intel Xeon X5650@2.67GHz
M/B	Supermicro X8DTG-D
Chipset	Intel 5520
RAM	12x4GB ECC@1333MHz
NIC	Myrinet 10G-PCIE-8A-C

Tab. 1: Technical specifications of the experimental testbed

The experiments are performed using Xen 4.3-unstable[3] and linux kernel version `3.7.1`. All cases except for Xen2MX, are set up using native Open-MX version `1.5.2`. For all tests we use `mx_pingpong`, a native MX micro-benchmark that sends and receives messages of variable size. We set the MTU to 9000 bytes. To capture CPU utilization numbers, we use `/proc/stat`.

4.2 Experiment 1: VM–to–Native

To accurately measure the overhead added to the communication data path, we measure the round-trip latency and ping-pong bandwidth between VMs and a native Linux host. We use `mx_pingpong` to transfer 10GB for 10000 iterations and present the results in Figures 10 and 11 respectively.

Figure 10 plots the round-trip latency achieved under the three aforementioned setups, as a func-

[3] changeset 26059:c1c549c4fe9e

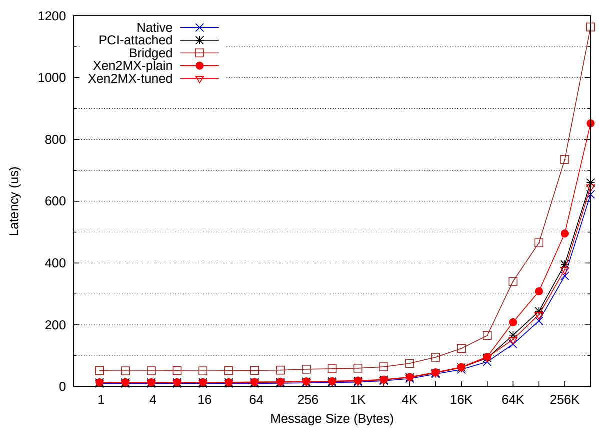


Fig. 10: RTT Latency between a single-core VM and a native linux host (lower is better).

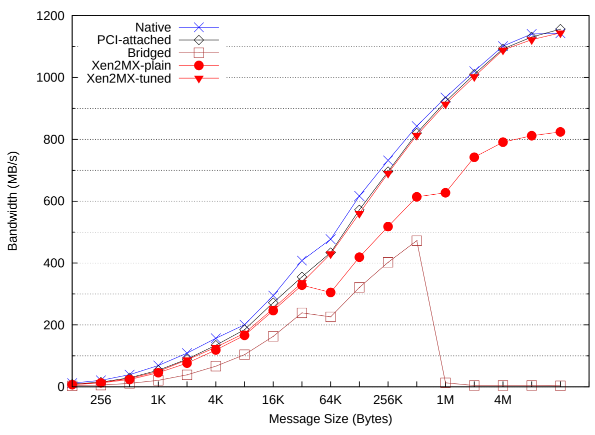


Fig. 11: Ping-pong bandwidth between a single-core VM and a native linux host (higher is better).

tion of message size. Our approach (`Xen2MX-plain`) is almost identical to the directly attached case (`PCI-attached`) for `SMALL` and `MEDIUM` messages whereas the `Bridged` case achieves 44μs. Using Xen2MX, transferring 1 byte across takes 14μs, less than half of the |Bridged— case and $\approx$ 1μs more than the best measured (`PCI-attached` case). An interesting observation is that after 4 KB, latency measurements start to ascend. Time rises proportionally to the message size after the MTU size (9000 bytes), where the 2-page limit granularity seriously affects latency. For `LARGE` messages (after 32 KB), the memory registration overhead is added to `Xen2MX-plain`, leading to increased latency compared to the `PCI-attached` case.

An important observation is that without the use of specialized hardware, Xen2MX is able to almost eliminate the latency overhead (reducing it to less than 1μs); this overhead ($\approx$ 4%) is attributed to the latency of the message that the frontend sends to the backend (roughly 200-300ns), and the dispatch logic of our design (300-400ns). As we cannot account for the first part that is Xen-related, we plan to optimize our dispatch logic and examine different approaches to eliminate even this small portion of latency.

Figure 11 shows the ping-pong bandwidth measured on the three aforementioned setups. `Xen2MX-plain` outperforms the `Bridged` case which exhibits unstable behavior. After 512 KB, performance drops to as low as 10 MB/s. 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 management duties at this rate. Debugging this specific shortcoming requires careful and extensive analysis; our educated guess is that this issue is due to the alloc / free cycle of shared pages that the netfront / netback drivers use. Performing a detailed latency breakdown to be able to optimize this path is one of our future plans.

`Xen2MX-plain` follows the scaling of the `PCI-attached` case until a certain message size (32 KB). This point is where memory registration becomes noticeable[4]. After the 32 KB threshold, `Xen2MX-plain` exhibits a slowdown, proportional to the message size, reaching 810 MB/s for 16 MB messages whereas the `PCI-attached` case saturates the link (1192 MB/s).

To track down this problem, we closely analyze the cost of the memory registration in order to gain in-

[4] Memory registration is not always on the critical path, depending on the implementation of the interconnect and the application needs.

sight into possible workarounds.

4.2.1 Cost of memory registration in message exchange

Figure 12 plots a breakdown of the time spent in a message exchange as a function of the message size for both the VM container and the guest. We consecutively send and receive 1GB of random data in 1000 iterations and capture the total amount of time spent in the most important parts of the stack. The first bar (`PLAIN`), plots the result obtained using the generic Xen2MX protocol; in the second one (`TUNED`), we enabled the registration cache (`MX_RCACHE=1`). The data volume is constant across different message sizes. However, the registration size varies, depending on the message size. For each iteration, the number of messages exchanged is obtained by $\frac{1GB}{message\_size}$, whereas the size of each registered region is equal to the message size. This test illustrates the overhead of memory registration for a given message size and demonstrates how we are able to overcome this shortcoming. For clarity, time is normalized to the `TUNED` case for each message size.

Both runs consume approximately the same time in sending and receiving data for a given message size. This is the time spent in the actual protocol processing and refers to the time required to transmit the relevant frames as well as the processing time for receiving and placing it in the user buffer. Regarding registration handling, we examine each case separately, in the following two sections.

`PLAIN` case

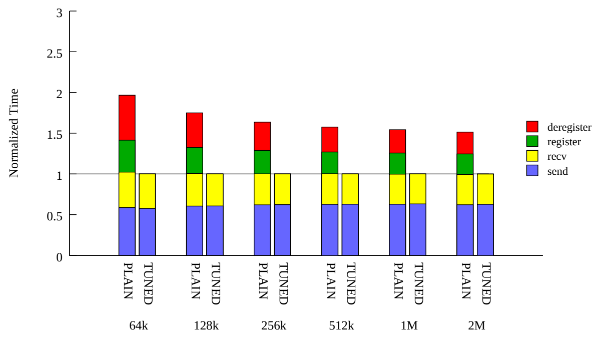
In the VM container, memory registration is almost as slow as the actual exchange for small messages (`PLAIN`); it only begins to decrease as we move to larger messages, reaching half of the data transfer for 2 MB messages. In the guest, however, the situation is on the opposite direction. For large messages, registration handling is up to 3 times as slow as the message exchange part. This is expected; sharing memory in Xen is done on a page granularity. For instance, when granting a user-space allocated memory region, the system has to track down the `mfn` for each one of the pages, claim grant references, and issue as many grant operations to the hypervisor as needed (for a 2 MB region, we need 512 grants). Xen offers the capability to batch grant requests from the backend (accepting a given number of grant references issued by the guest). Thus, for small messages, memory registration is slow in the backend, whereas for large messages, the guest becomes the bottleneck. In both cases, the grant handling the dominant part of memory registration, reducing the total achievable bandwidth up to 30% (810 MB/s vs. 1192 MB/s of the `PCI-attached` case).

`TUNED` case

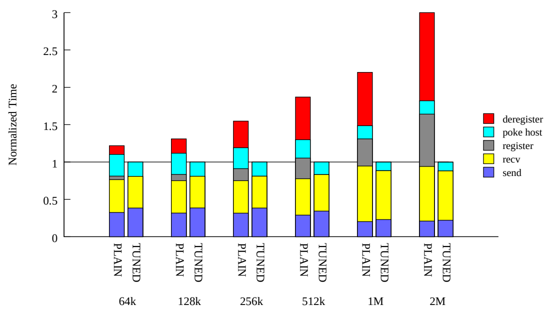
As discussed in Section 3.3, high-performance interconnection protocols require memory registration. On top of this overhead, in a VM environment, registration comes with the burden of granting, which is quite an expensive operation. To work around this limitation, we enable the registration cache; this allows subsequent message transfers to queue send/receive operations without paying any additional registration or granting costs. This approach to registration assumes that the application will not tamper with malloc or other memory handling functions, and will often reuse buffers, in order to amortize the high cost of a single up front memory registration operation. For many applications this is a reasonable assumption.

Thus, as expected, when we enable the cache option, the registration time disappears (Figure 12, right bars). The Open-MX library keeps track of preregistered regions and instead of allocating space and granting every single time all the relevant pages, it just re-uses existing, granted memory regions. This has a direct effect on the latency and bandwidth measurements presented above (Figures 10 and 11).

Selecting the `MX_RCACHE` option, enables Xen2MX to scale *exactly* as the `PCI-attached` case, even after the 32 KB threshold, for both latency and bandwidth measurements. Xen2MX almost eliminates the latency overheads due to intermediate virtualization layers (Figure 10, `Xen2MX-tuned`) and nearly saturates the 10Gbps link (Figure 11, `Xen2MX-tuned`) without the need to use specialized adapters or in-



(a) VM container



(b) Guest VM

Fig. 12: Breakdown of the time spent in memory registration, guest–to–host notification, and data exchange as a function of the message size. Time is normalized to the `TUNED` case for each message size.

flexible setups such as the directly attached case.

4.3 Experiment 2: VM–to–VM

The second experiment consists of two separate parts: the first one is measuring the aggregate bandwidth over Xen2MX, between a variable number of VMs that exchange messages, residing in different VM containers; the second part is capturing the effect of our protocol to the CPU utilization of the system, compared to the bridged case. This way, we can assess specific trade-offs one is obliged to take in order to choose between performance in terms of network throughput and CPU load. In what follows, we enable memory registration caching, so `Xen2MX` refers to `Xen2MX-tuned`. Figures 13, 14 and 15 plot our findings.

Figure 13 presents the aggregate throughput of the first VM container when transferring 256 KB messages as a function of the number of VMs per container participating in the experiment. The bars plot the throughput whereas the lines show the CPU utilization (%) for the VM container.

`Xen2MX` outperforms the `Bridged` case for 256 KB messages in all settings (1 to 8 VMs), being 20% faster for 8 VMs. CPU utilization for the `Xen2MX` case increases linearly to the number of VMs per container, as opposed to the `Bridged` case which consumes almost the same CPU time regardless of the number of VMs spawned.

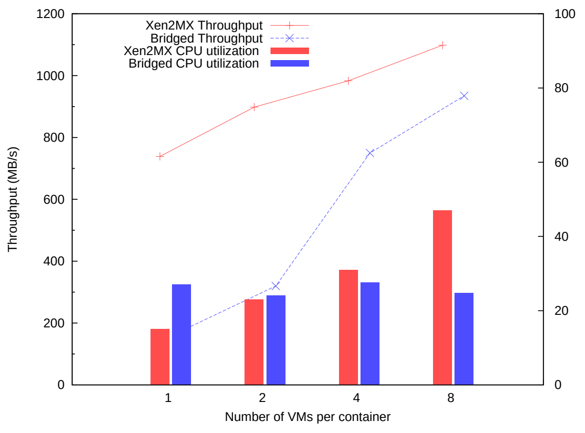


Fig. 13: CPU utilization and Throughput as a function of the number of VMs per container. We setup two VM containers, spawn up to 16 single-core VMs on each and deploy `mx_pingpong` to capture the total bandwidth and CPU utilization the system achieves. The bars show the aggregate bandwidth as measured by the VM container (left for `Xen2MX`, right for `Bridged`). We keep 4 cores for each driver domain and plot the CPU utilization that we observe (squares for `Xen2MX` and circles for `Bridged`).

However, one of the main differences between

Xen2MX and the Bridged case is that the latter does most of the protocol processing in the VM, while the former consumes more time in the backend. To validate this hypothesis, we gather CPU utilization measurements from each one of the VMs and plot the average on top of the VM container's CPU utilization (Figure 14). The dark bars show the VM container's CPU utilization while the light bars plot the VMs' CPU utilization[5].

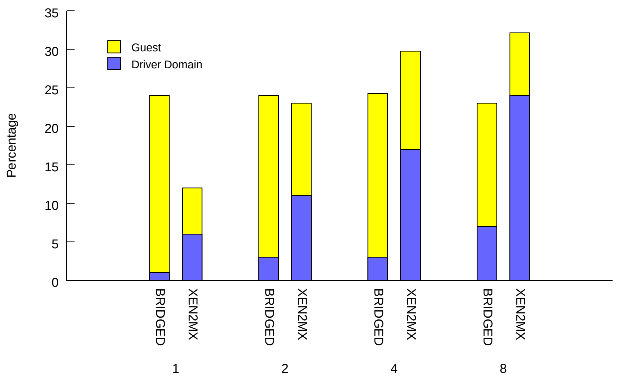


Fig. 14: Normalized CPU utilization for both driver domain and guests as a function of the number of VMs per container. The dark bars plot the CPU utilization we captured in the VM container whereas the light bars show the relevant measurement in the VMs (average).

Indeed, the CPU time consumed in the VM for Xen2MX is less than half of the one for the Bridged case (light bars). For a single VM, Xen2MX consumes approximately half of the total CPU resources compared to the generic case. For 8 VMs, our approach consumes 8% more CPU time than the Bridged case. However, Xen2MX outperforms the default case in terms of both bandwidth and latency for all message resizes and any number of VMs.

To gain further insight into where time is spent regarding CPU resources, we present the time of the most CPU intensive parts of the stack for the VM (Figure 15). These numbers are gathered using the following method: we transfer 1G through mx_pingpong back and forth, across all 8 VMs using 128 KB messages for 1000 iterations. We then observe for a given time period all CPU time metrics provided by the OS in the guest, as well as the aggregate bandwidth as measured by the network interface in the driver domain. We plot the average value of each setting for all VMs. As a result, we are able to qualitatively assess the degree of impact to the total CPU utilization of both cases, Xen2MX and the Bridged one.

[5] Both ratios are relevant to each separate machine; for example, a 100% for a dual core VM container is two times more than a 100% for a single core VM.

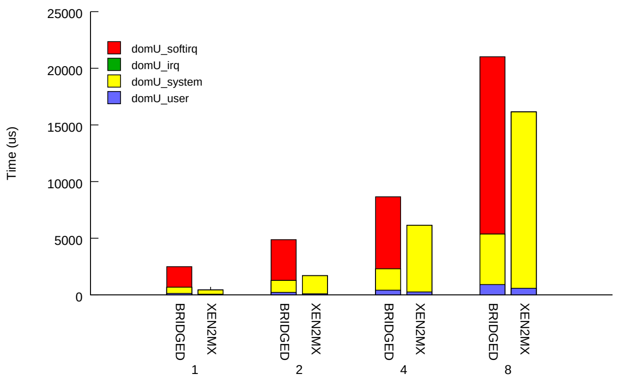


Fig. 15: VM CPU time breakdown as a function of the number of VMs per container.

Xen2MX CPU time is almost dominated by system time, whereas in the Bridged case, most of the time is consumed in softirq. An interesting remark that validates our previous findings is that our approach consumes significantly less time than the Bridged case, reaching approximately 20% less for 8 VMs (≈ 5ms for the above experiment). The reason for this behavior is that we are able to handle operations semantically, in an interconnect context (send operations with large chunks of data, batching requests to the backend and so on), as opposed to the Bridged case which just processes Ethernet frames (in an MTU granularity). However, due to our backend implementation, the total CPU utilization measurements (driver domain and guest) is still approximately 8% more than in the Bridged case. This is a fairly straightforward trade-off one is obliged to take if high-performance communication in a VM environment is the ultimate goal.

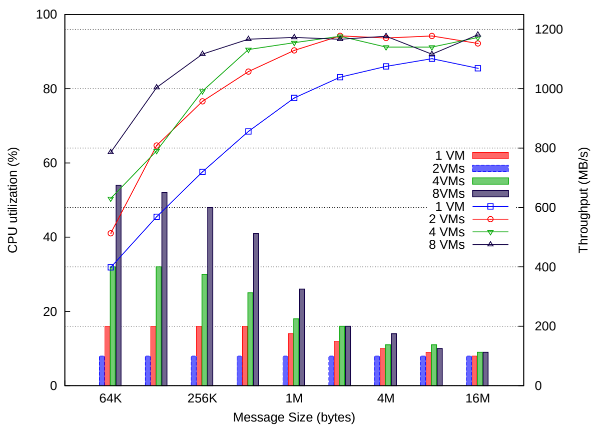


Fig. 16: Throughput and driver domain CPU utilization as a function of the message size and the number of VMs per container. We spawn up to 16 single-core VMs and capture the total, aggregate bandwidth of all cases (1, 2, 4 and 8 VMs per container) as measured by the VMs. At the same time, we observe the CPU utilization of the VM container. The VM container has 4 cores while all vCPUs are unpinned – the decision where to place the vCPUs is left to the Xen scheduler.

To elaborate more on the scaling factor of Xen2MX as we add more VMs, we measure the CPU utilization and total throughput as a function of the message size and the number of VMs per container. Figure 16 plots our findings.

We spawn up to 8 single-core VMs and measure the aggregate bandwidth of all cases (1, 2, 4 and 8 VMs) as a function of the message size. At the same time, we observe the CPU utilization of the VM container, to which we assign 4 cores. This figure validates our previous measurements and shows that CPU utilization increases proportionally to the number of VMs but decreases with larger messages. For smaller message sizes ($<$ 256 KB messages), the aggregate bandwidth increases proportionally to the number of VMs; Xen2MX begins to almost saturate the link from 256 KB messages when 8 VMs put pressure on the system ($\approx$ 55% CPU utilization).

For messages $>$ 512 KB, Xen2MX saturates the link while at the same time, CPU utilization decreases, reaching $\approx$ 9% for 8VMs exchanging 16 MB messages.

Another interesting point is that while we add more VMs to the system, Xen2MX lowers the threshold for link saturation; for instance when 1 VM is running, the total throughput is $\approx$ 850 MB/sec for 512 KB messages whereas when 8 VMs are running Xen2MX achieves near line-rate bandwidth.

4.4 Scaling factor

In this section we evaluate how `Xen2MX` scales with a large number of VMs and compare bandwidth and CPU utilization results to the `Bridged` case. We spawn up to 20 VMs in each VM container (up to 40 VMs total) and massively overwhelm the system with send and receive loads, simulating a real-life scenario.

We enable hyperthreading to fully exploit our testbed's capabilities (Table 1), and choose not to assign each vCPU to a physical core. The driver domain is assigned 4 cores and, as we observed, the Xen scheduler makes sure that the vCPUs of the VMs and the driver domain do not co-exist on the same physical core for each run. The setup is as in Experiment 2 (Section 4.3), but instead of 16 VMs, we spawn 40 (VM_1...VM_{20} on $Host_0$ and VM_{21}...VM_{40} on $Host_1$). Data exchange is realized between opposite VMs (VM_1 to VM_{21}, VM_2 to VM_{22} etc.).

The bars present the CPU utilization of each case for the VM container, whereas the points show the aggregate throughput as observed from the VM container. We validated these findings with the numbers returned from each VM separately.

Using this setting, we repeat Experiment 2. Figure 17 illustrates the following: The `Bridged` case exhibits unstable behavior after 512 KB as previously. `Xen2MX` outperforms the `Bridged` case in terms of throughput for all message sizes (64 KB – 16 MB). CPU utilization decreases for large messages reaching 18% for 16 MB, while almost saturating the link. This validates our statement that Xen2MX scales efficiently.

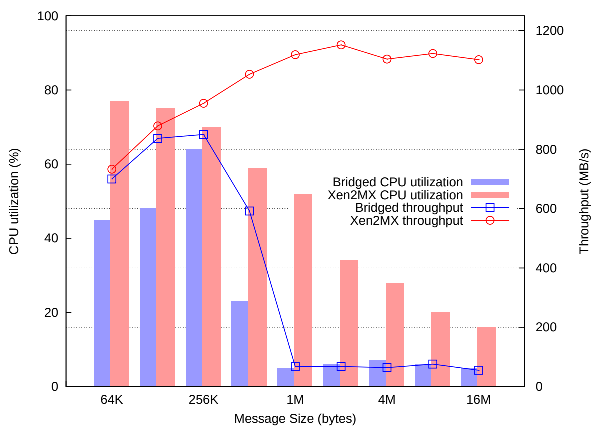


Fig. 17: Throughput and driver domain CPU utilization while overwhelming the system. We spawn 20 single-core VMs on each host (40 total) and deploy Experiment 2 on this setup. We capture the total, aggregate bandwidth as measured by the VM container and, at the same time, observe the CPU utilization of the system for both cases `Bridged` and `Xen2MX`. We enable hyperthreading to obtain all 24 threads (2 threads per core) and leave all vCPUs unpinned.

5 Related Work

The advent of 10G Ethernet and its extensive use in cluster interconnects has given rise to a large body of literature on optimizing upper-level protocols, specifically, protocol handling and processing overheads [32, 13, 9, 34]. Several optimizations have been introduced: OS-bypass communication, zero-copy for TCP/IP, offloading the network stack on a separate, dedicated core etc. Based on these findings, an efficient application-to-NIC data path is achieved, exposing HPC interconnect semantics to applications using generic hardware, such as 10G Ethernet adapters. We propose the integration of high-performance cluster interconnect semantics to virtualization environments, building on this strand of the literature.

Recent advances in virtualization technology have minimized overheads associated with CPU or memory sharing [39, 40, 41], leading to applications achieving near-native computational performance when deployed in VM environments. However, I/O is a completely different story: intermediate virtualization layers impose significant overheads when multiple VMs share network or storage devices [20, 22, 33].

Previous work on this limitation has mainly focused on PV. Menon et al. [20] propose optimizations of the Xen network architecture by adding new capabilities to the virtualized network interface (scatter/gather I/O, TCP/IP checksum offload, TCP segmentation offload). Ram et al. [28] enhance the grant mechanism, while Dong et al. [6] propose the extension of the VMM scheduler for real-time response support. The authors in [33] and [29] present memory-wise optimizations to the Xen networking architecture. [4] proposes optimizations to the TCP/IP stack, while [17] introduces a framework to encapsulate virtual network traffic for KVM [15]. While all the aforementioned optimizations appear ideal for server-oriented workloads, the TCP/IP stack imposes a significant overhead when used for message passing.

Contrary to the previous approaches, Liu et al. [19] describe VMM-bypass I/O over Infiniband. Their approach is novel and based on Xen's split driver model. In [23], the authors present the design of a similar framework using Myrinet interfaces. Many features presented in these studies can be used by modern virtualization platforms to overcome the bandwidth and latency limitations imposed by software overheads. We build on this idea, but instead of providing a virtualized device driver for a cluster interconnect architecture, we develop a paravirtual framework that integrates HPC communication semantics into the virtualization platform.

Research has also focused on optimizing intra-node, inter-VM communication: in [5], the authors present a shared memory communication library used for intra-node message passing for the KVM and achieve near-native performance in terms of MPI bandwidth. Huang et al. [12] design an inter-VM, intra-node communication library, implement it on top of a popular MPI library and evaluate its performance. They show that a combination of a VM-aware MPI library and VMM-bypass data-paths for inter-node communication, imposes very little overhead on the execution of HPC applications in VM environments.

To achieve direct communication between hardware and guest domains some works have examined hardware optimizations: Raj et al. [27] describe the creation of self-virtualized devices, while Mansley et al. [14] propose the use of accelerators for device accesses. Dong et al. [7] present the integration of SR-IOV techniques into the Xen platform; their results indicate that domains can achieve isolated and secure direct access to the hardware. Finally, Auernhammer et al. [3] propose architectural improvements to CPUs, to provide support for virtual interfaces. They simulate an Infiniband-like NIC to examine the implications of doorbell mechanisms and address translations in the send path.

Apart from hardware approaches, many studies have attempted to optimize the integration of IOV into popular virtualization platforms; Gordon et al. [10] study the interrupt overhead of VM traffic in KVM and present an approach where exitless interrupts lead to considerable improvements in network throughput. This work is complementary to [38], where the authors optimize the direct assignment technique to account for efficient DMA remapping.

However, these approaches have two major drawbacks: (a) since these techniques are mainly based on hardware to ensure high bandwidth, the number of VMs that can share such devices is bound by hardware constraints; (b) they present limited or no flexibility, one of virtualization's major advantages; hence, migration becomes more difficult due to heterogeneity issues between data centers or internal availability zones. Additionally, decoupling the host from managing VM's access to the network renders intra-VM communication handling more difficult. VMs that coexist in the same container either have to explicitly inform the host about their communication needs, or the hardware must be smart enough to be able to inform the host and build an in-house channel of communication between them.

To optimize paravirtual network setups, Gordon et al. [11] present specific improvements to the KVM stack in order to minimize the inter-core communication overhead imposed by guest-to-host and host-to-guest notifications. However, their work is focused on TCP/IP workloads and do not present comparable results to our approach; for instance, they use 1 core for the host, which gets saturated at $\approx$ 390 MB/sec. Their approach could be combined with Xen2MX in order to minimize the CPU utilization overheads due to guest–host communication.

Despite IOV's clear performance benefits, recent works on efficient paravirtual high-performance communication validate our trajectory. Ranadive et al. [30] present a software-based approach which is similar to VMM-bypass I/O [19]; they develop a paravirtual interface that provides an RDMA-like interface for VMware ESXi guests. Although their design is feature–complete, performance numbers have not been reported yet.

6 Discussion

The literature suggests that paravirtualized approaches are the de-facto standard for network I/O in generic cloud infrastructures. Our results provide a clear indication that Xen2MX could become a viable alternative to generic Ethernet virtual interfaces in an HPCs context. In this section, we present possible uses of Xen2MX and highlight specific optimizations that could be employed on our framework:

- *Optimize message handling.* At the moment, Xen2MX handles incoming messages via the control I/O ring (described in Section 3). Employing alternative approaches to guest-to-host kicks, could eliminate the latency overhead, provided that the hypervisor supports enhanced notification mechanisms.
- *Efficient page sharing.* A more elaborate mechanism for page sharing could boost Xen2MX's performance for memory registrations, in favor of disabling the registration cache. For instance, many applications could benefit from using different page sizes (2 MB vs. 4 KB), while, at the same time, memory registration time would be reduced significantly.
- *Optimize the receive path CPU time consumption by utilizing DMA off-load engines.* Al-

though we have developed the option to offload receive copies to the DMA engines engines present on our system, it seems that further debugging and testing is needed to obtain maximum performance benefits. We would like to closely examine the data path and provide an extended Xen2MX version where CPU utilization will be reduced to a minimum and the placing of receive frames to the relevant buffers can be efficiently overlapped with other operations (i.e. sending more data, application computation on the physical cores etc.).

- *Evaluate the effect of latency sensitive scheduling techniques.* To account for real-time processing in cloud environments, we need to rethink standard scheduling techniques for efficient vCPU-to-physical core mappings. Based on recent studies [1, 31], evaluating Xen2MX using the CPU pool concept (adopted in the credit scheduler, Xen v4.2), as well as NUMA-aware VM placing would illustrate trade-offs associated with high-performance communication and efficient consolidation in cloud environments.
- *Extend Xen2MX to utilize multiple driver domains.* Moreover, the deployment of Xen2MX to more than a single driver domain could improve the consolidation factor. We expect that our two-part memory registration procedure, combined with the high-availability consumer-producer scheme, will provide the same performance results with the minimum CPU utilization on the host.

Our design is generic enough to be applicable to all paravirtualized approaches, providing an easy, scalable way to deploy communication-intensive applications in the Cloud using commodity, off-the-shelf components. Although the current implementation is based on Xen, we plan to explore porting the design to other hypervisors, such as KVM. Essentially, the upper– and lower–lever parts of the protocol will remain intact, whereas the guest-to-host communication will be adapted using KVM's API (guest–host notification mechanism, memory sharing etc.).

We also plan to follow closely the trends of Ethernet-based message passing protocols such as the Common Communication Interface (CCI [2]) and render our framework compatible with such protocols.

The major challenges in the adoption of virtualization solutions in HPC workloads are the elimination of I/O overheads as well as the efficient scaling of VMs within a VM container [35]. Our work focuses on addressing these challenges. With regards to the former, Xen2MX achieves near-native performance (within 4% of the native case) without the use of specialized adapters at the cost of no more than 8% CPU utilization overhead compared to the bridged case. Considering scalability, Xen2MX is able to efficiently handle a large number of concurrent VMs running on a host, providing line saturation even for smaller message sizes.

7 Conclusion

We have presented the design and implementation of Xen2MX, a framework for high-performance communication in commodity cloud infrastructures. Xen2MX is based on the paravirtualization concept, consisting of a backend and a frontend driver running on the host and guest respectively. Our implementation is based on Open-MX, exploiting its features of binary compatibility with Myrinet/MX and wire compatibility with MXoE. Xen2MX is open-source software, available online at `https://github.com/ananos/xen2mx`.

Experimental evaluation of our prototype has revealed the following: (a) Xen2MX is able to achieve near-native performance, outperforming bridged setups, the generic case of networking in virtualized environments, adding less than 4% overhead compared to directly attached or native setups in terms of throughput and latency. (b) Our framework adds $\approx$ 8% of CPU utilization overhead to the host system while saturating the network link compared to the bridged setup which is capped to $\approx$ 850MB/sec. (c) Xen2MX scales efficiently with large numbers of VMs, sustaining line-rate bandwidth for even 20 single-core VMs exchanging 256KB messages with the network.

References

[1] Aragiorgis, D., Nanos, A., Koziris, N.: Coexisting scheduling policies boosting i/o virtual machines. In: Euro-Par 2011: Parallel Processing Workshops, *Lecture Notes in Computer Science*, vol. 7156, pp. 407–415. Springer Berlin Heidelberg (2012). DOI 10.1007/978-3-642-29740-3_46

[2] 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)

[3] Auernhammer, F., Sagmeister, P.: Architectural support for user-level network interfaces in heavily virtualized systems. In: WIOV'10: Proceedings of the 2nd conference on I/O virtualization, pp. 7–7. USENIX, Berkeley, CA, USA (2010)

[4] Ben-Yehuda, M., Breitgand, D., Factor, M., Kolodner, H., Kravtsov, V., Pelleg, D.: NAP: a building block for remediating performance bottlenecks via black box network analysis. In: ICAC '09: Proc. of the 6th Intern. Conference on Autonomic computing, pp. 179–188. ACM, NY, USA (2009). DOI 10.1145/1555228.1555270

[5] Diakhaté, F., Pérache, M., Namyst, R., Jourdren, H.: Efficient shared memory message passing for inter-vm communications. In: 4th Workshop on Virtualization in High-Performance Cloud Computing (VHPC '08), Euro-par 2008 (2008)

[6] Dong, Y., Dai, J., Huang, Z., Guan, H., Tian, K., Jiang, Y.: Towards high-quality I/O virtualization. In: SYSTOR '09: Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference, pp. 1–8. ACM, NY, USA (2009). DOI 10.1145/1534530.1534547

[7] Dong, Y., Yu, Z., Rose, G.: SR-IOV networking in Xen: architecture, design and implementation. In: WIOV'08: Proceedings of the First conference on I/O virtualization, pp. 10–10. USENIX, Berkeley, CA, USA (2008)

[8] Geoffray, P.: Opiom: off-processor io with myrinet. In: Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on, pp. 261 –268 (2001). DOI 10.1109/ccgrid.2001.923202

[9] Goglin, B.: High-Performance Message Passing over generic Ethernet Hardware with Open-MX. Parallel Computing **37**(2), 85–100 (2011). URL http://hal.inria.fr/inria-00533058. Open-MX

[10] Gordon, A., Amit, N., Har'El, N., Ben-Yehuda, M., Landau, A., Tsafrir, D., Schuster, A.: ELI: Bare-Metal Performance for I/O Virtualization. In: ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS) (2012)

[11] Gordon, A., Har'El, N., Landau, A., Ben-Yehuda, M., Traeger, A.: Towards Exitless and Efficient Paravirtual I/O. In: The 5th Annual International Systems and Storage Conference (SYSTOR) (2012)

[12] Huang, W., Koop, M.J., Gao, Q., Panda, D.K.: Virtual machine aware communication libraries for high performance computing. In: SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing. ACM, NY, USA (2007). DOI 10.1145/1362622.1362635

[13] Karlsson, S., Passas, S., Kotsis, G., Bilas, A.: MultiEdge: An Edge-based Communication Subsystem for Scalable Commodity Servers. p. 28. IEEE Computer Society, Los Alamitos, CA, USA (2007). DOI 10.1109/ipdps.2007.370218

[14] Kieran Mansley Greg Law, D.R.: Getting 10 gb/s from xen: Safe and fast device access from unprivileged domains. In: Euro-Par 2007 Workshops: Parallel Processing (2007)

[15] Kivity, Avi: KVM: the Linux Virtual Machine Monitor. In: OLS '07: The 2007 Ottawa Linux Symposium, pp. 225–230 (2007)

[16] Koukis, E., Nanos, A., Koziris, N.: Gm-block: Optimizing data movement in a block-level storage sharing system over myrinet. Cluster Computing **13**, 349–372 (2010). DOI 10.1007/s10586-009-0106-y

[17] Landau, A., Hadas, D., Ben-Yehuda, M.: Plugging the hypervisor abstraction leaks caused by virtual networking. In: SYSTOR '10: Proceedings of the 3rd Annual Haifa Experimental Systems Conference, pp. 1–9. ACM, NY, USA (2010). DOI 10.1145/1815695.1815716

[18] Liu, J., Huang, W., Abali, B., K. Panda, D.: High performance vmm-bypass i/o in virtual machines. In: ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference, pp. 3–3. USENIX Association, Berkeley, CA, USA (2006)

[19] Liu, J., Huang, W., Abali, B., Panda, D.K.: High performance VMM-bypass I/O in virtual machines. In: ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference, pp. 3–3. USENIX, Berkeley, CA, USA (2006)

[20] Menon, A., Cox, A.L., Zwaenepoel, W.: Optimizing network virtualization in Xen. In: ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference, pp. 2–2. USENIX, Berkeley, CA, USA (2006)

[21] Myricom: Myrinet eXpress (MX): A High Performance, Low-Level, Message-Passing Interface for Myrinet (2006). Http://www.myri.com/scs/MX/doc/mx.pdf

[22] Nanos, A., Goumas, G., Koziris, N.: Exploring I/O virtualization data paths for MPI applications in a cluster of VMs: a networking perspective. In: 5th Workshop on Virtualization in High-Performance Cloud Computing (VHPC '10). Ischia - Naples, Italy (2010)

[23] 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)

[24] Nanos, A., Koziris, N.: Xen2MX: towards high-performance communication in the cloud. In: 7th Workshop on Virtualization in High-Performance Cloud Computing (VHPC '12). Rhodes Island, Greece (2012)

[25] Nanos, A., Nikoleris, N., Psomadakis, S., Kozyri, E., Koziris, N.: A Smart HPC Interconnect for Clusters of Virtual Machines. In: 6th Workshop on Virtualization in High-Performance Cloud Computing (VHPC '11). Bordeaux,France (2011)

[26] PCI SIG: SR-IOV specification (2007). Http://www.pcisig.com/specifications/iov/single_root/

[27] Raj, H., Schwan, K.: High performance and scalable I/O virtualization via self-virtualized devices. In: HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing, pp. 179–188. ACM, NY, USA (2007). DOI 10.1145/1272366.1272390

[28] Ram, K.K., Santos, J.R., Turner, Y.: Redesigning xen's memory sharing mechanism for safe and efficient I/O virtualization. In: WIOV'10: Proceedings of the 2nd conference on I/O virtualization, pp. 1–1. USENIX, Berkeley, CA, USA (2010)

[29] Ram, K.K., Santos, J.R., Turner, Y., Cox, A.L., Rixner, S.: Achieving 10 Gb/s using safe and transparent network interface virtualization. In: VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, pp. 61–70. ACM, NY, USA (2009). DOI 10.1145/1508293.1508303

[30] Ranadive, A., Davda, B.: Toward a Paravirtual vRDMA Device for VMware ESXi Guests. VMware Technical Journal, Winter 2012 **1**(2) (2012)

[31] Ranadive, A., Gavrilovska, A., Schwan, K.: Resourceexchange: Latency-aware scheduling in virtualized environments with high performance fabrics. In: Proceedings of the 2011 IEEE International Conference on Cluster Computing, CLUSTER '11, pp. 45–53. IEEE Computer Society, Washington, DC, USA (2011). DOI 10.1109/cluster.2011.14

[32] Recio, R., Culley, P., Garcia, D., Hilland, J.: An RDMA Protocol Specification (Version 1.0) This document is a Release Specification of the RDMA Consortium. (2002)

[33] Santos, J.R., Turner, Y., Janakiraman, G., Pratt, I.: Bridging the gap between software and hardware techniques for I/O virtualization. In: ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference, pp. 29–42. USENIX, Berkeley, CA, USA (2008)

[34] Shalev, L., Satran, J., Borovik, E., Ben-Yehuda, M.: IsoStack—Highly Efficient Network Processing on Dedicated Cores. In: USENIX ATC '10: USENIX Annual Technical Conference (2010)

[35] Simons, J.E., Buell, J.: Virtualizing high performance computing. SIGOPS Oper. Syst. Rev. **44**(4), 136–145 (2010). DOI 10.1145/1899928.1899946

[36] Whitaker, A., Shaw, M., Gribble, S.D.: Denali: Lightweight virtual machines for distributed and networked applications. In: In Proc. of the USENIX Annual Technical Conference (2002)

[37] Yassour, B.A., Ben-Yehuda, M., Wasserman, O.: Direct device assignment for untrusted fully-virtualized virtual machines. Technical Report H-0263, IBM Research (2008)

[38] Yassour, B.A., Ben-Yehuda, M., Wasserman, O.: On the dma mapping problem in direct device assignment. In: SYSTOR 2010: The 3rd Annual Haifa Experimental Systems Conference (2010)

[39] Youseff, L., Seymour, K., You, H., Zagorodnov, D., Dongarra, J., Wolski, R.: Paravirtualization effect on single- and multi-threaded memory-intensive linear algebra software. Cluster Computing **12**, 101–122 (2009). DOI 10.1007/s10586-009-0080-4

[40] Youseff, L., Wolski, R., Gorda, B., Krintz, R.: Paravirtualization for HPC Systems. In: In Proc. Workshop on Xen in High-Performance Cluster and Grid Computing, pp. 474–486. Springer (2006)

[41] Zhong, A., Jin, H., Wu, S., Shi, X., Gao, W.: Performance implications of non-uniform vcpu-pcpu mapping in virtualization environment. Cluster Computing pp. 1–12 (2012). DOI 10.1007/s10586-012-0199-6