MyriXen: Message Passing in Xen Virtual Machines over Myrinet and Ethernet

Anastassios Nanos and Nectarios Koziris

National Technical University of Athens,
School of Electrical and Computer Engineering,
Computing Systems Laboratory,
Zografou Campus, Zografou 15780, Greece
{ananos,nkoziris}@cslab.ece.ntua.gr

Abstract. Data access in HPC infrastructures is realized via user-level networking and OS-bypass techniques through which nodes can communicate with high bandwidth and low-latency. Virtualizing physical components requires hardware-aided software hypervisors to control I/O device access. As a result, line-rate bandwidth or lower latency message exchange over 10GbE interconnects hosted in Cloud Computing infrastructures can only be achieved by alleviating software overheads imposed by the Virtualization abstraction layers, namely the VMM and the driver domains which hold direct access to I/O devices.

In this paper, we present MyriXen, a framework in which Virtual Machines efficiently share network I/O devices bypassing overheads imposed by the VMM or the driver domains. MyriXen permits VMs to optimally exchange messages with the network via a high performance NIC, leaving security and isolation issues to the Virtualization layers. Smart Myri-10G NICs provide hardware abstractions that facilitate the integration of the MX semantics in the Xen split driver model. With MyriXen, multiple VMs exchange messages using the MX message passing protocol over Myri-10G interfaces as if the NIC was assigned solely to them. We believe that MyriXen can integrate message passing based applications in clusters of VMs provided by Cloud Computing infrastructures with near-native performance.

Keywords: Virtualization, Xen, Myrinet, Ethernet, MyriXen, Myri-10g, Linux, I/O, DMA, Virtualized I/O, Message Passing, MX.

1 Introduction

Current Cloud Computing research is focused on providing a scalable, on-demand, clustered computing environment. One of the major challenges in this field is bridging the gap between Virtualization techniques and high performance network I/O retrieval techniques [1]. To meet the I/O needs of HPC applications running in Virtualization environments, research has focused on alleviating overheads that arise due to intermediate software layers. Hardware vendors [2] have become increasingly aware of this issue and provide the community with smart I/O devices that can export multiple interface instances using software [3] or hardware [4,5].

H.X. Lin et al. (Eds): Euro-Par 2009 Workshops, LNCS 6043, pp. 395–403, 2010.
© Springer-Verlag Berlin Heidelberg 2010

Integrating I/O Virtualization semantics in HPC infrastructures can both facilitate research and offer software management freedom without affecting isolated application execution. Having a cluster of VMs that is almost identical to a cluster of workstations and can be set up in hours or minutes seems quite intriguing.

While HPC interconnects provide abstractions that can be exploited in Virtual Machine execution environments, they lack architectural support. In this paper, we describe MyriXen, a framework in which Virtual Machines share network I/O devices efficiently bypassing overheads imposed by the VMM or the driver domain model. Specifically, MyriXen allows VMs to optimally exchange messages with the network via a high performance NIC leaving only security and isolation issues to be handled by the hypervisor and the NIC itself. In the following sections, we present some background information followed by MyriXen's design architecture.

2 Background

In Virtualization environments, the basic building blocks of the system (i.e. CPUs, memory and I/O devices) are multiplexed by the Virtual Machine Monitor (VMM). The latter may allow VMs to access these resources directly, in order to maximize performance. Xen [6] consists of the hypervisor, the driver domains and the VMs (guest domains). Driver domains are privileged guests that access I/O devices directly and provide the VMs abstractions to interface with the hardware via a *split driver model*. Driver domains host a *backend* driver while VM kernels host a *frontend* driver exposing a generic device API to guest kernels or userspace. The frontend communicates with the backend via an event channel communication mechanism along with interrupt routing, page–flipping, and shared memory techniques.

In Xen, memory is virtualized in order to provide physically contiguous regions to Operating Systems running on guest domains. This is achieved by adding a a per-domain memory abstraction called *pseudo-physical* memory. Therefore, in Xen, *machine memory* refers to the physical memory of the entire system, whereas pseudo-physical memory refers to the memory regions exported to the Operating Systems running in each guest domain.

Xen Paravirtualized Network I/O. Xen's paravirtualized (PV) network architecture is based on a split driver model. Guest VMs host the *netfront* driver which exports a generic Ethernet API to kernelspace. The driver domain, which directly accesses network hardware, hosts the hardware specific driver, the protocol interface driver, and the *netback* driver. The latter communicates with netfront via a dedicated event channel. Upon initialization of a guest domain, the netfront binds to the netback driver, which in turn binds to a dummy network interface bridged with the physical network interface in software. Thus, network packets originating from the VM are transfered (copied or flipped) to the netback driver and are injected to the NIC via the software bridge.

Contrary to the previous approach, a Xen guest domain can directly access an I/O device at the expense of system's security. For example, suppose a VM accesses directly a generic Ethernet NIC. To achieve maximum bandwidth the VM kernel allocates memory for building an Ethernet frame and, with the help of the VMM, informs the NIC's DMA engine about the physical address of the buffer. The NIC then DMAs data from the VM's space to its packet buffers and emits the frame to the network. However, the DMA transfer begins without checking the validity of the source or destination, which evidently raises security issues.

Xen Grant Mechanism. To efficiently share memory pages across guest domains, Xen exports a *grant* mechanism to guest domains. Xen's grants are stored in *grant tables* and provide a generic mechanism to memory sharing between domains. Network I/O device drivers are based on this mechanism in order to exchange control information and data via shared memory. Each domain has its own grant table.

Myrinet/MX basics. In order to fully comprehend MyriXen's design, it is of utmost importance to present the basic structure of Myrinet/Myrinet eXpress (MX) [7]. To run MX in Virtualization environments, these features have to be integrated into Xen's device driver architecture.

Myrinet [8] is a low-latency, high bandwidth interconnection infrastructure for clusters. Two generations of Myrinet are currently available: Myrinet-2000 and Myri-10G. Myrinet achieves low-latency cut-through switching using source routing. Myri-10G is based on the same physical layer as 10GbE and can either use the source-routed Myrinet protocol or 10GbE as the Data Link layer.

Myrinet NICs feature a RISC microprocessor called Lanai, which consists of: the CPU, the copy engine, two packet interfaces, each with its own on-chip send and receive packet buffers, a Z-port (XAUI Myri-10G / 10G Ethernet), and a PCI Express port. The Local Bus (LBUS) is an interface to a fast, synchronous, static SRAM.

To reduce the overhead of OS involvement, Myrinet employs user-level networking techniques. In this model, an application process is allowed to control the Network Interface (NI) directly; since the OS is no longer invoked for communication, its role is undertaken by a combination of application-level libraries and firmware executing on the NIC, while data exchange between the two is set up by privileged code inside an OS kernel module.

To provide user-level networking facilities to applications, the MX message passing system is used. An application is granted control of the NI by mapping part of the NI memory space into its own virtual memory. User-level communication is accomplished by using unprivileged load/store instructions to the relevant VM segments bypassing OS abstractions and copies. This is a privileged operation, which is done via system calls to the MX kernel module during the application's initialization phase. Each of these parts, called MX endpoints, acts as an isolated virtual network interface at the process level for the application and contains an unprotected part, which is mapped to userspace, and a protected,

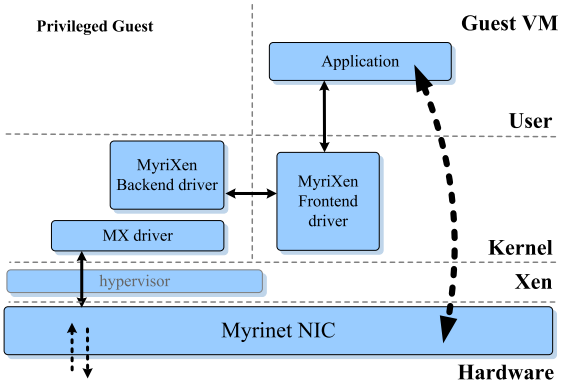


Fig. 1. MyriXen

trusted part, which is only accessible by the kernel module and the firmware. An endpoint provides an entry point to the interconnect's hardware, protected from other processes, with fairness relative to the other endpoints opened on the same NIC.

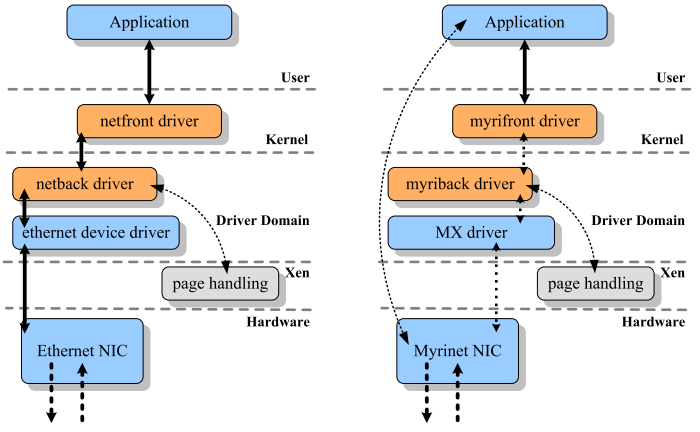
3 MyriXen

Like most device drivers, the MX driver cannot export multiple interfaces for a single Myrinet NIC. Thus, a split driver model approach is required for multiple Xen VMs to share a single NIC. In the following sections we describe our prototype design.

Architecture. Fig 1 illustrates the basic design architecture of MyriXen. The backend runs on top of the native MX driver in the driver domain. It waits for incoming requests from the frontend drivers running in the VMs. The frontend driver replaces the core MX driver and, once loaded, establishes two event channels with the backend driver: the first one is used to process requests initiated from the VM; the second one is used for informing the guest domain about MX events.

The frontend driver is a relatively thin layer and exports to VM userspace the same API as the core MX driver. One of its tasks is the installation of mappings for the MX library to access the NIC directly. Moreover, its role is to provide the mechanism to open and close MX endpoints via the event channel mechanism. The backend driver sets up the Myrinet NIC to support these features and issues acknowledgments for event completion or error.

The split driver model used in Xen poses difficulties for user-level direct NIC access in Xen VMs. To enable driver domain bypass techniques, we need to let VMs have direct access to certain NIC resources. The building block of MyriXen is *myriback* which allows *myrifront* to communicate with the MX core driver, and thus, install the prerequisites to send or receive a message to / from the network.



(a) Xen PV software stack (b) MyriXen software stack

Fig. 2. Xen and MyriXen network I/O Software Stack

The myrifront driver, similarly to the netfront driver, communicates with the backend via an event channel mechanism. Contrary to the netfront / netback architecture, MyriXen utilizes the backend in conjunction with the core MX driver to grant pages to the VM user space and install mappings that can simulate the normal case; the netfront driver, on the other hand, uses these channels to send or receive packets (as a data path). Fig. 2(a) shows Xen PV net I/O software stack as opposed to MyriXen's software stack in Fig. 2(b).

MyriXen Semantics. To communicate with the network, an application running in a VM needs privileged access to the NIC. In order to provide isolated, virtualized access to the Myri-10G NIC we must take into account the following issues:

Initialization: For applications to communicate using MX, the MX library along with the NIC have to be initialized. The preparation steps needed for initialization include allocating basic structures for the library to communicate with the Lanai. This is essentially a mapping operation: the VMs forward requests to the myriback driver and wait for completion; the backend driver executes the operation and provides the acknowledgment back to the frontend. The frontend manages these resources using handles that are provided by the backend via the event channel mechanism.

Endpoint management: Message passing over the network occurs between endpoints. Thus, a VM has to open and close an endpoint before and after communication takes place, respectively. Opening an MX endpoint means obtaining a specific handle from the Lanai and the MX core driver. This is realized in two steps. During the first step, the frontend requests an endpoint. The backend gets informed via the relevant event channel and requests a free endpoint from the NIC. The NIC returns a handle to that endpoint. The second step consists of the acknowledgment sent back to the frontend along with the handle.

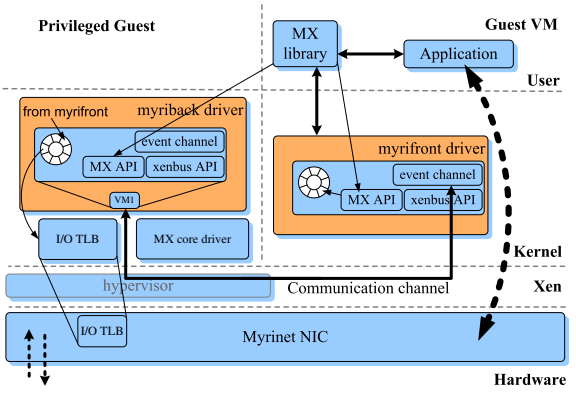


Fig. 3. MyriXen network I/O architecture

Memory registration: After the Initialization and the Endpoint opening phase, the frontend has to register memory regions that will participate in the message exchange. The memory registration requests originate from the frontend and provide grant references to the Xen hypervisor. The MX core driver registers these memory regions to the Lanai, and forms an on-chip page table cache. This is a necessary step in order to realize zero-copy data transfer.

Message Matching: MX offers a rich message matching interface to applications and application libraries in order to alleviate matching overhead imposed by the kernel or generally OS involvement. Thus, MyriXen can exploit the existing MX semantics for matching to achieve direct delivery of messages to VMs bypassing receive-path copies that impose significant overhead.

Protection: As mentioned in section 2, Xen uses a grant mechanism to transfer pages from privileged to guest domains and vice-versa. MX is based on zero-copy and OS-bypass techniques to achieve line-rate bandwidth for 10Gbps networks. However, these techniques can lead to significant throughput degradation due to Xen's architecture. To avoid illegitimate access to arbitrary memory regions, an I/O TLB cache mechanism can be used to transfer valid page tables to the Lanai SRAM. Thus, memory protection is guarantied by having the Lanai check for valid memory regions before programming its DMA engines.

Address Translation: DMA transfers between userspace buffers and Lanai's on-chip packet buffers are invoked by applications that have already opened an endpoint and have registered specific memory regions. The registration process involves pinning pages that will be used for DMA transfers as well as informing the MX core driver and the NIC about these pages. Address translation in Xen consists of two steps: the first is a virt-to-phys translation on the VM side (myrifront) and the second is a pseudo-phys-to-machine translation done be the hypervisor so as to end up with addresses that the Lanai DMA engines are able to follow.

Discussion. Note that initialization, endpoint management, and memory registration are steps that occur outside the critical path of network intensive applications. Although these operations are resource intensive (PIO to the NIC, asynchronous events, grant references, etc.), they take place before or after the communication phase, and thus, their performance impact is not significant.

In MyriXen, data flows directly from applications to the NIC leaving only control issues to be handled by the hypervisor or the driver domain. Applications control the DMA engines of the Lanai chip directly using load/store instructions to NIC memory mapped regions.

An important feature of our design is the decoupling of data transfers from the Virtualization layers. The implications of this mechanism for the overall throughput constitute a possible caveat of our approach. Specifically, the way the control path interferes with data communication may result in significant overhead, which needs to be examined with extensive performance evaluation using microbenchmarks.

Finally, one could also criticize how MyriXen handles isolation: MX provides isolated access by using MX endpoints exporting a virtual NI to any application that requests access to the network. Our design of MyriXen is such that MX semantics to applications remain unaltered, so the isolation features provided by MX would also characterize MyriXen.

4 Related Work

Previous work has concluded that the integration of Virtualization semantics in specialized software running on Network Processors can isolate and finally minimize the hypervisor and driver domain overheard associated with device access. Liu et al. [9] describe VMM-bypass I/O using the Infiniband architecture. Their approach is novel and based on Xen's split driver model. Many features presented in this work can be used by modern Virtualization platforms to overcome the bandwidth and latency limitations imposed by software overheads. Although their model is thoroughly designed, they focus on Infiniband, which uses specialized hardware without open specifications.

Mansley et al. [10] developed a safe, direct data path between the VM and the network using Solarflare Ethernet NICs. However, their approach is device specific and cannot be used for efficient message passing, since existing messaging protocols have to be stacked above Ethernet. Our approach accounts for this problem and aims to keep MX's full compatibility with the applications.

Santos et al. [1] present a performance analysis of network device I/O in Xen and identify possible bottlenecks. They also propose optimizations and implement a small subset of them. The primary objective of this work is to present a generic framework for sharing smart NICs in Xen VMs. As an extension to [1], in [3], the authors describe mechanisms to overcome the bandwidth limitations imposed by the VMM and the driver domain model in Xen and provide an efficient and direct data path for VMs to access the network. Unfortunately, the authors do not focus on HPC environments although the features they propose could be exploited by message passing protocols.

While the aforementioned studies present a promising framework they have only been implemented on top of Ethernet ignoring the potential advantages of Myrinet. This study aspires to provide insight into integrating existing messaging protocol semantics to Virtualization platforms.

5 Conclusions and Future Work

We have described the basic design of MyriXen, a thin split driver layer on top of the Myri-10G MX driver to support message passing in Xen VMs over the wire protocols supported in Myri-10G infrastructures. Message passing occurs in a direct I/O data path leaving only control and management issues to the driver domains and the VMM. MyriXen's design is based on the Xen split driver model in order to sustain manageable infrastructures that can provide VMs with security, isolation, and migration capabilities even in heterogeneous hardware configurations.

MPI applications over a cluster of VMs are liable to significant performance degradation due to limited network performance [11]. This is mainly caused by overheads imposed by the driver domains, which directly access I/O devices. MyriXen accounts for this problem by combining the following two important features: it utilizes the Xen split driver model and, at the same time, installs a direct application-to-NIC data path for message exchange.

We believe this approach is a step towards integrating HPC applications in Virtualization environments, such as Cloud Computing infrastructures. MyriXen provides VM-level networking semantics in an already deployed Virtualization platform, Xen, with a vast user base. Our future work will be firmly oriented towards evaluating our prototype design and presenting an extensive performance evaluation of MyriXen in conjunction with latency breakdown analysis on MPI applications running on clusters of VMs. We believe that there is scope for these applications to benefit greatly from MyriXen's direct data path and achieve performance close to native.

References

1. Santos, J.R., Turner, Y., Janakiraman, G., Pratt, I.A.: 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 Association, Berkeley (2008)
2. Abramson, D., Jackson, J., Muthrasanallur, S., Neiger, G., Regnier, G., Sankaran, R., Schoinas, I., Uhlig, R., Vembu, B., Wiegert, J.: Intel Virtualization Technology for Directed I/O. Intel Technology Journal 10(03), 179–192 (2006)
3. 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, New York (2009)

4. 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, New York (2007)
5. LeVasseur, J., Panayappan, R., Skoglund, E., du Toit, C., Lynch, L., Ward, A., Rao, D., Neugebauer, R., McAuley, D.: Standardized But Flexible I/O for Self-Virtualizing Devices. In: Ben-Yehuda, M., Cox, A.L., Rixner, S. (eds.) Workshop on I/O Virtualization. USENIX Association (2008)
6. 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: Proceedings of the nineteenth ACM symposium on Operating systems principles, pp. 164–177. ACM, New York (2003)
7. Myricom: Myrinet eXpress (MX): A High Performance, Low-Level, Message-Passing Interface for Myrinet (2006), `http://www.myri.com/scs/MX/doc/mx.pdf`
8. Boden, N.J., Cohen, D., Felderman, R.E., Kulawik, A.E., Seitz, C.L., Seizovic, J.N., Su, W.: Myrinet: A Gigabit-per-Second Local Area Network. IEEE Micro 15(1), 29–36 (1995)
9. Liu, J., Huang, W., Abali, B., Panda, K.: High performance VMM-bypass I/O in virtual machines. In: ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference, p. 3. USENIX Association, Berkeley (2006)
10. Mansley, K., Law, G., Riddoch, D.: Getting 10 Gb/s from Xen: Safe and Fast Device Access from Unprivileged Domains. In: Euro-Par 2007 Workshops: Parallel Processing (2007)
11. Youseff, L., Wolski, R., Gorda, B., Krintz, C.: Evaluating the Performance Impact of Xen on MPI and Process Execution For HPC Systems. In: First International Workshop on Virtualization Technology in Distributed Computing, VTDC 2006 (2006)