A Smart HPC interconnect for clusters of Virtual Machines

Anastassios Nanos[1]*, Nikos Nikoleris[2]**, Stratos Psomadakis[1], Elisavet Kozyri[3]***, and Nectarios Koziris[1]

[1] Computing Systems Laboratory, National Technical University of Athens
[2] Uppsala Architecture Research Team, Uppsala University
[3] Cornell University

Abstract. In this paper, we present the design of a VM-aware, high-performance cluster interconnect architecture over 10Gbps Ethernet. Our framework provides a direct data path to the NIC for applications that run on VMs, leaving non-critical paths (such as control) to be handled by intermediate virtualization layers. As a result, we are able to multiplex and prioritize network access per VM. We evaluate our design via a prototype implementation that integrates RDMA semantics into the privileged guest of the Xen virtualization platform. Our framework allows VMs to communicate with the network using a simple user-level RDMA protocol. Preliminary results show that our prototype achieves 681MiB/sec over generic 10GbE hardware and relieves the guest from CPU overheads, while limiting the guest's CPU utilisation to 34%.

1 Introduction

Nowadays, Cloud Computing infrastructures provide flexibility, dedicated execution, and isolation to a vast number of services. These infrastructures, built on clusters of multicores, offer huge processing power; this feature makes them ideal for mass deployment of compute-intensive applications. However, I/O operations in virtualized environments are usually handled by software layers within the hypervisor. These mechanisms multiply the numerous data paths and complicate the way data flow from applications to the network.

In the HPC world, applications utilize adaptive layers to overcome limitations that operating systems impose in order to ensure security, isolation, as well as fairness in resource allocation and usage. To avoid the overhead associated with user-to-kernel–space communication, cluster interconnects adopt a user-level networking approach. However, when

* {ananos,psomas,nkoziris}@cslab.ntua.gr
** nikos.nikoleris@it.uu.se
*** ekozyri@cs.cornell.edu

applications access I/O devices without regulation techniques, security issues arise and hardware requirements increase. Currently, only a subset of the aforementioned layers is implemented in virtualization platforms.

In this paper, we propose a framework capable of providing VMs with HPC interconnect semantics. We examine the implications of bypassing common network stacks and explore direct data paths to the NIC. Our approach takes advantage of features found in cluster interconnects in order to decouple unnecessary protocol processing overheads from guest VMs and driver domains. To evaluate our design, we develop a lightweight RDMA protocol over 10G Ethernet and integrate it in the Xen virtualization platform. Using network microbenchmarks, we quantify the performance of our prototype. Preliminary results indicate that our implementation achieves 681MiB/sec with negligible CPU involvement on the guest side, while limiting CPU utilization on the privileged guest to 34%.

2 Background and Related Work

In virtualization environments, the basic building blocks of the system (i.e. CPUs and memory) are multiplexed by the Virtual Machine Monitor (VMM). In ParaVirtualized (PV) [1] VMs, only privileged instructions are trapped into the VMM; unprivileged operations are carried out directly on hardware. Since this is the common case for HPC applications, nearly all overheads from intermediate virtualization layers in an HPC context are associated with I/O and memory management. Data access is handled by privileged guests called *driver domains* that help VMs interact with the hardware via a *split driver model*. Driver domains, host a *backend* driver, while guest VM kernels host *frontend* drivers, exposing a per-device class API to guest user– or kernel–space. With SR/MR-IOV [2] VMs exchange data with the network using a direct VM-to-NIC data path provided by a combination of hardware and software techniques: thus, device access by multiple VMs is multiplexed in firmware running on the hardware itself, bypassing the VMM on the critical path.

Overview of the Xen Architecture Xen [3] is a popular VMM that uses PV. It consists of a small hypervisor, driver domains, and the VMs (guest domains). Xen Memory Management: In Xen, memory is virtualized in order to provide contiguous regions to OS's running on guest domains. This is achieved by adding a per-domain memory abstraction called *pseudo-physical* memory. So, in Xen, *machine memory* refers to the physical memory of the entire system whereas pseudo-physical memory refers to the physical memory that the OS in any guest domain is aware of. Xen PV Network I/O: Xen's PV network architecture is based on a split driver model. Guest VMs host the *netfront* driver, which exports a generic Ethernet API to kernel-space. The driver domain hosts the hardware specific driver and the *netback* driver, which communicates with the frontend via an event channel mechanism and injects frames to the NIC via a software bridge. Xen Communication Mechanisms: As communication between the frontend and the backend is a major part of PV, we briefly describe Xen's doorbell mechanisms. *Grant Mechanism:* To efficiently share pages across guest domains, Xen exports a *grant* mechanism. Xen's grants are stored in *grant tables* and provide a generic mechanism for memory sharing between domains. This subsystem allows shared-memory communications between unprivileged domains and can be used to create ring buffers between them. *Event Channels:* Two guests can initialize an event channel between them and then exchange events that trigger the execution of the corresponding handlers.

High-performance Interconnects: Typical HPC applications utilize mechanisms to overcome limitations imposed by general purpose operating systems. These layers are usually: (a) communication libraries (MPI), (b) mechanisms that bypass OS kernels to optimize process scheduling and device access (user-level networking, zero-copy, page-cache bypass, etc.). High-performance communication protocols comprise the backend layers of popular parallel programming frameworks (e.g. MPI). These protocols run on adapters that export part of the network interface to user–space via *endpoints*. 10G Ethernet: While VMs can communicate with the network over TCP, UDP, or even IP protocol layers, this choice entails unwanted protocol processing. In VM environments, HPC protocol messages are encapsulated into TCP/IP datagrams, so significant latency ensues.

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 [4–7]. Recent advances in virtualization technology have minimized overheads associated with CPU or memory sharing. However, I/O is a completely different story: intermediate virtualization layers impose significant overheads when multiple VMs share network or storage devices [8, 9]. Previous work on this limitation has mainly focused on PV. Menon et al. [10] 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). [11] enhances the grant mechanism, while [12] proposes the extension of the VMM scheduler for real-time response support. The authors in [13] and [14] present memory-wise optimizations to the Xen networking architecture. While all the aforementioned optimizations appear ideal for server-oriented workloads, the TCP/IP stack imposes a significant overhead when used for a message passing library, which is standard practice in HPC applications. Contrary to the previous approaches, Liu et al. [15] describe VMM-bypass I/O over Infiniband. Their approach is novel and based on Xen's split driver model. In [16], the authors present the design of a similar framework using Myrinet interfaces. We build on this idea, but instead of providing a virtualized device driver for a cluster interconnect architecture, we develop a framework that forwards requests from the VM's application space to the native device driver.

3 Design and Implementation

Our approach is built on the following pillars: (a) a library which provides an application interface to guest's user-space; (b) a frontend that forwards guest's applications requests to lower-level virtualization layers; (c) a backend that multiplexes requests to access the network.

Main Components The user-space library exports the basic API which defines the primitive operations of our protocol. Processes issue commands via their endpoints (see section 2), monitor the endpoints' status and so on.

The API defines functions to handle control messages for opening / closing an endpoint, memory registration and RDMA read / write. The library is also responsible for informing the frontend to setup the communication channel with the backend. These primitive operations can be used to implement higher level communication substacks, such as MPI or shared memory libraries. Our approach exports basic RDMA semantics to VM's user-space using the following operations:

Initialization: The guest side of our framework is responsible for setting up an initial communication path between the application and the backend. Frontend-Backend communication: This is achieved by utilizing the messaging mechanism between the VM and the backend. This serves as a means for applications to instruct the backend to transmit or wait for communication, and for the backend to inform the guest and the applications of error conditions or completion events. We implemented this mechanism using event channels and grant references. Export interface instance to user-space: To support this type of mechanism we utilize endpoint semantics. The guest side provides operations to *open* and *close* endpoints, in terms of allocating or deallocating and memory mapping control structures residing on the backend.

Memory registration: In order to perform RDMA operations from user-space buffers, applications have to inform the kernel to exclude these buffers from memory handing / relocation operations. To transfer data from application buffers to the network, the backend needs to access memory areas. This happens as follows: the frontend pins the memory pages, grants them to the backend and the latter accepts the grant in order to gain access to these pages. An I/O Translation Look-aside Buffer (IOTLB) is used to cache the translations of pages that will take part in communication. This approach ensures the validity of source and destination buffers, while enabling secure and isolated access multiplexing. Guest-to-Network: The backend performs a look-up in the IOTLB, finds the relevant machine address and informs the NIC to program its DMA engines to start the transfer from the guest's memory. The DMA transfer is performed directly to the NIC and as a result, packets are encapsulated into Ethernet frames, before being transmitted to the network. We use a zero-copy technique on the send path in order to avoid extra, unnecessary copies. Packet headers are filled in the backend and the relevant (granted) pages are attached to the socket buffer. Network-to-Guest: When an Ethernet frame is received from the network, the backend invokes the associated packet handler. The destination virtual address and endpoint are defined in the header so the backend performs a look-up on its IOTLB and is performs the necessary operations.Data are then copied (or DMA'd) to the relevant (already registered) destination pages.

Wire protocol: Our protocol's packets are encapsulated into Ethernet frames containing the type of the protocol (a unique type), source and destination MAC addresses.

Data Movement: Figure 1 shows the data paths either for control or data movement: *Proposed approach:* Applications issue requests for RDMA operations through endpoints. The frontend passes requests to the backend using the event channel mechanism (dashed arrow, b_1). The backend performs the necessary operation, either registering memory buffers (filling up the IOTLB), or issuing transmit requests to the Ethernet driver (dashed arrow, c_1). The driver, then, informs the NIC to DMA data from application to the on-chip buffers (dashed arrow, d_1). *Ideal approach:* Although the proposed approach relaxes the system from processing and context-switch overheads, ideally, VMs could communicate directly with the hardware, lowering the multiplexing authority to the NIC's firmware (solid arrows).

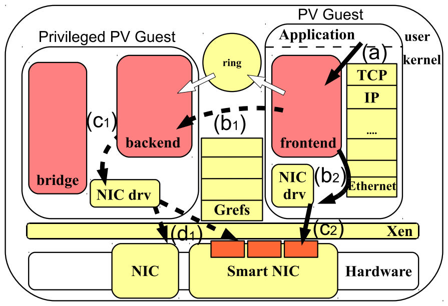


Fig. 1.

4 Performance Evaluation

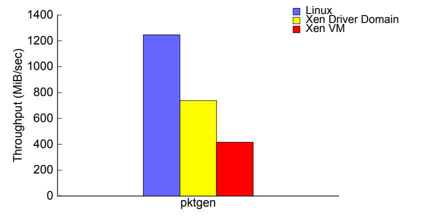
We use a custom synthetic microbenchmark to evaluate our approach over our interconnect sending unidirectional RDMA write requests. To obtain a baseline measurement, we implement our microbenchmark using TCP sockets. TCP/IP results were verified using netperf [17] in `TCP_STREAM` mode and varying message sizes. As a testbed, we used two Quad core Intel Xeon 2.4GHz with an Intel 5500 chipset, with 4GB main memory. The network adapters used are two PCIe-4x Myricom 10G-PCIE-8A 10GbE NICsin Ethernet mode, connected back-to-back. We used Xen version 4.1-unstable and Linux kernel version 2.6.32.24-pvops both for the privileged guest and the VMs. The MTU was set to 9000 for all tests. We use 1GB memory for each VM and 2GB for the privileged guest. CPU utilization results are obtained from `/proc/stat`. To eliminate Linux and Xen scheduler effects we pinned all vCPUs to physical CPUs and assigned 1 core per VM and 2 cores for the privileged guest, distributing interrupt affinity to each physical core for event channels and the Myrinet NICs In the following, `TCP_SOCK` refers to the TCP/IP network stack and `ZERO_COPY` refers to our proposed framework.

4.1 Results

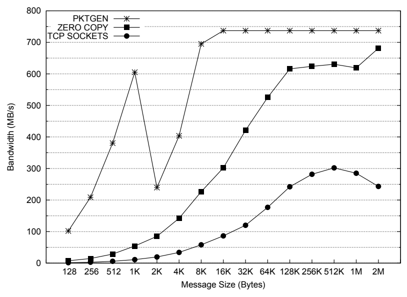
To obtain a baseline for our experiments, we run the pktgen utility of the Linux kernel. This benchmark uses raw Ethernet and, thus, this is the upper bound of all approaches. Figure 2(a) plots the maximum achievable socket buffer production rate when executed in vanilla Linux (first bar), inside the Privileged Guest (second bar) and in the VM (third bar). Clearly, the PVops Linux kernel encounters some issues with Ethernet performance, since the privileged guest can achieve only 59% of the vanilla Linux case. As mentioned in Section 2, Xen VMs are offered a virtual Ethernet device via the netfront driver. Unfortunately, in the default configuration, this device does not feature specific optimizations or accelerators and, as a result, its performance is limited to 416MiB/sec (56% of the PVops case)[4].

Bandwidth and Latency: Figure 2(b) plots the aggregate throughput of the system over TCP/IP (filled circles) and over our framework (filled squares) versus the message size. We also plot the Driver domain's pktgen performance as a reference. For small messages (<4KB) our framework outperforms TCP by a factor of 4.3 whereas for medium-sized messages

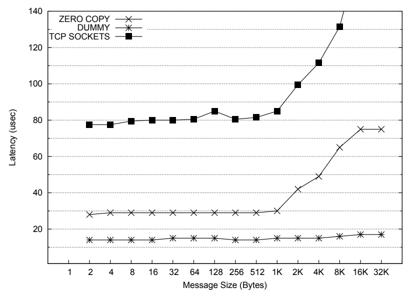
[4] for details on raw Ethernet performance in Xen PVops kernel see http://lists.xensource.com/archives/html/xen-users/2010-04/msg00577.html



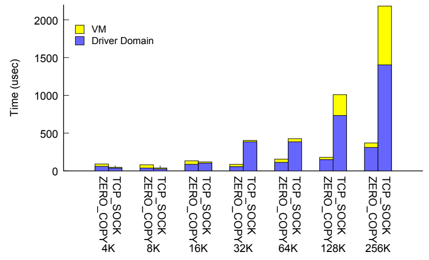
(a) Maximum achievable socket buffer production rate



(b) Aggregate bandwidth



(c) One-way Latency



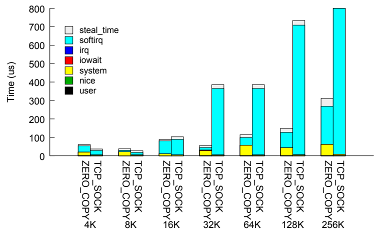
(d) Aggregate CPU time vs. RDMA message size (send and receive path)

Fig. 2.

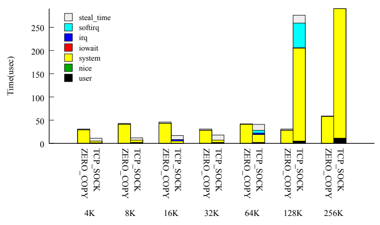
(i.e. 128KB) by a factor of 3. For large messages (>512K) our framework achieves nearly 92% of the pktgen case (for 2MB messages) and is nearly 3 times better than the TCP approach. The suboptimal performance of the microbenchmark over TCP is due mainly to: (a) the complicated protocol stack (TCP/IP) (see Section 4.1) and (b) the unoptimized virtual Ethernet interface of Xen.

From a latency point of view (Figure 2(c)), an RDMA message over TCP sockets takes 77μsec to cross the network, whereas over our framework it takes 28μsec. To set a baseline latency-wise, we performed a `DUMMY` RDMA write: 1 byte originating from an application inside the VM gets copied to the privileged guest, but instead of transmitting it to the network, we copy it to another VM on the same VM container. Results from this test show that 14μsecs are needed for 1 byte to traverse the intermediate virtualization layers.

CPU time for RDMA writes: In the HPC world, nodes participating in clusters except for low-latency and high-bandwidth communication, require computational power.



(a) CPU time breakdown for the driver domain



(b) CPU time breakdown for the VM (send and receive path)

Fig. 3. CPU time breakdown for both the driver domain and the guests

Our approach bypasses the TCP/IP stack; we assume that, in this case, the CPU utilization of the system is relaxed. In order to validate this assumption we examine the CPU time spent in both approaches. We measure the total CPU time when two VMs perform RDMA writes of varying message sizes over the network (TCP and ZERO_COPY approach). In Figure 2(d), we plot the CPU time both for the driver domain and the VM. It is clearly shown that for 4K to 32K messages the CPU time of our framework is constant, as opposed to the TCP case where CPU time increases proportionally to the message size. When the 64K boundary is crossed, TCP CPU time increases by an exponential factor due to intermediate switches and copies both on the VM and the driver domain. Our framework is able to sustain low CPU time on the Privileged Guest and almost negligible CPU time on the VM. To further investigate the sources of CPU time consumption, we plot the CPU time breakdown for the Privileged Guest and the VM in Figures 3(a) and 3(b), respectively.

In the driver domain (Figure 3(a)): **(a)** Our framework consumes more CPU time than the TCP case for 4KB and 8KB messages. This is due to the fact that we use zero-copy only on the send side; on the receive side, we have to copy data from the socket buffer provided by the NIC to pages originating from the VM. **(b)** For messages larger than 32KB, our approach consumes at most 30% CPU time of the TCP case, reaching 15% (56 vs. 386) for 32K messages. **(c)** In our approach, system time is non-negligible and varying from 20% to 50% of the total CPU time spent in the Privileged Guest. This is due to the fact that we haven't yet implemented page swapping on the receive path. In the VM (Figure 3(b)): **(d)** Our approach consumes constant CPU time for almost all message sizes (varying from 30μsecs to 60μsecs). This constant time is due to the way the application communicates with the frontend (IOCTLs). However, in the TCP case, for messages larger than 64K, CPU time increases significantly. This is expected, as all the protocol processing (TCP/IP) is done inside the VM. Clearly, system time is almost 60% of the total VM CPU time for 256K messages reaching 75% for 128K. **(e)** Our approach exhibits negligible softirq time (apparent mostly in the receive path). This is due to the fact that the privileged guest is responsible for placing data originating from the network to pages we have already pinned and granted. On the other hand, the TCP case consumes softirq time as data elevate on the TCP/IP network stack to reach the application's socket.

5 Conclusions

We have described the design and implementation of a VM-aware high-performance cluster interconnect architecture. Our approach integrates HPC interconnect semantics in PV VMs using the split driver model. Specifically, we build a framework that consists of a low-level backend driver running in the driver domain, a frontend running in the VMs, and a user-space library that provides applications with our protocol semantics. We implement these RDMA semantics using a lightweight protocol and deploy network microbenchmarks to evaluate its performance.

Our work extends the concept of user-level networking to VM-level networking. Allowing VMs to interact with the network without the intervention of unoptimized virtual Ethernet interfaces or the TCP/IP stack, yields significant performance improvements in terms of CPU utilization and throughput.

Our prototype implementation supports generic 10GbE adapters in the Xen virtualization platform. Experimental evaluation leads to the following two remarkable results: our framework sustains 92% (681MiB/sec over 737MiB/sec) of the maximum Ethernet rate achieved in our system; at this maximum attainable performance, the driver domain's CPU utilization is limited to 34%, while the guest's CPU is idle.

We are confident that our approach is generic enough to be applicable to various virtualization platforms. Although our work is focused on PV systems, it can be easily extended by decoupling the proposed lightweight network stack from the driver domain to dedicated guests or hardware. This way, virtualization can gain considerable leverage in HPC application deployment from a networking perspective. We plan to enrich our protocol semantics in order to implement low-level backends for higher-level parallel frameworks such as MPI or MapReduce.

References

1. Whitaker, A., Shaw, M., Gribble, S.D.: Denali: Lightweight virtual machines for distributed and networked applications. In: In Proc. of the USENIX Annual Tech-

nical Conference. (2002)

2. PCI SIG: SR-IOV (2007) http://www.pcisig.com/specifications/iov/single_root/.
3. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I.A., Warfie ld, A.: Xen and the Art of Virtualization. In: SOSP '03: Proc. of the 19th ACM symposium on Operating systems principles, NY, USA, ACM (2003) 164–177
4. 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.
5. Karlsson, S., Passas, S., Kotsis, G., Bilas, A.: MultiEdge: An Edge-based Communication Subsystem for Scalable Commodity Servers, Los Alamitos, CA, USA, IEEE Computer Society (2007) 28
6. Goglin, B.: Design and Implementation of Open-MX: High-Performance Message Passing over generic Ethernet hardware. In: CAC 2008: Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2008, Miami, FL, IEEE Computer Society Press (April 2008)
7. 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)
8. Youseff, L., Wolski, R., Gorda, B., Krintz, C.: Evaluating the Performance Impact of Xen on MPI and Process Execution For HPC Systems. In: 1st Intern. Workshop on Virtualization Techn. in Dstrb. Computing. VTDC 2006.
9. 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), Naples-Ischia, Italy (2010)
10. 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, Berkeley, CA, USA, USENIX (2006) 2–2
11. 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, Berkeley, CA, USA, USENIX (2010) 1–1
12. 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, NY, USA, ACM (2009) 1–8
13. 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, Berkeley, CA, USA, USENIX (2008) 29–42
14. 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, NY, USA, ACM (2009) 61–70
15. 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, Berkeley, CA, USA, USENIX (2006) 3–3
16. 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)
17. Jones, R.: Netperf http://www.netperf.org.