Towards an HPC I/O Framework for Clusters of VMs

Anastassios Nanos and Nectarios Koziris {ananos,nkoziris}@cslab.ece.ntua.gr

Motivation

Virtualization is about:

(a) Multiplexing CPU/Memory subsystems: PV handles this issue properly (unprivileged instructions are executed directly)

(b) Scheduling of VM workloads: In an HPC context, we assign each vCPU to a physical core

(c) Multiplexing – Managing I/O access

Contribution

We propose a framework for integrating HPC I/O semantics in Virtual Environments. Our approach builds on common ParaVirtualization concepts and introduces a new device class: HPC I/O devices. We focus on the split driver model and present a frontend/backend mechanism that forwards specific API requests to the native HPC device driver. To improve performance, we eliminate unnecessary data copies by remapping pages that take place in the data movement process. Our approach is applicable to any kind of device, ranging from network adapters to hardware accelerators.

Open Challenges

Higher-level parallel frameworks: Our protocol implements simple RDMA semantics (READ, WRITE). We are in the process of extending its capabilities to support higher-level frameworks for application parallelism such as MPI or MapReduce.

Performance: Efficient I/O device sharing and support in Virtualized environments does not appear to be sufficiently mature. Even in full virtualization setups, many I/O devices have to be exported as emulated devices to VMs.

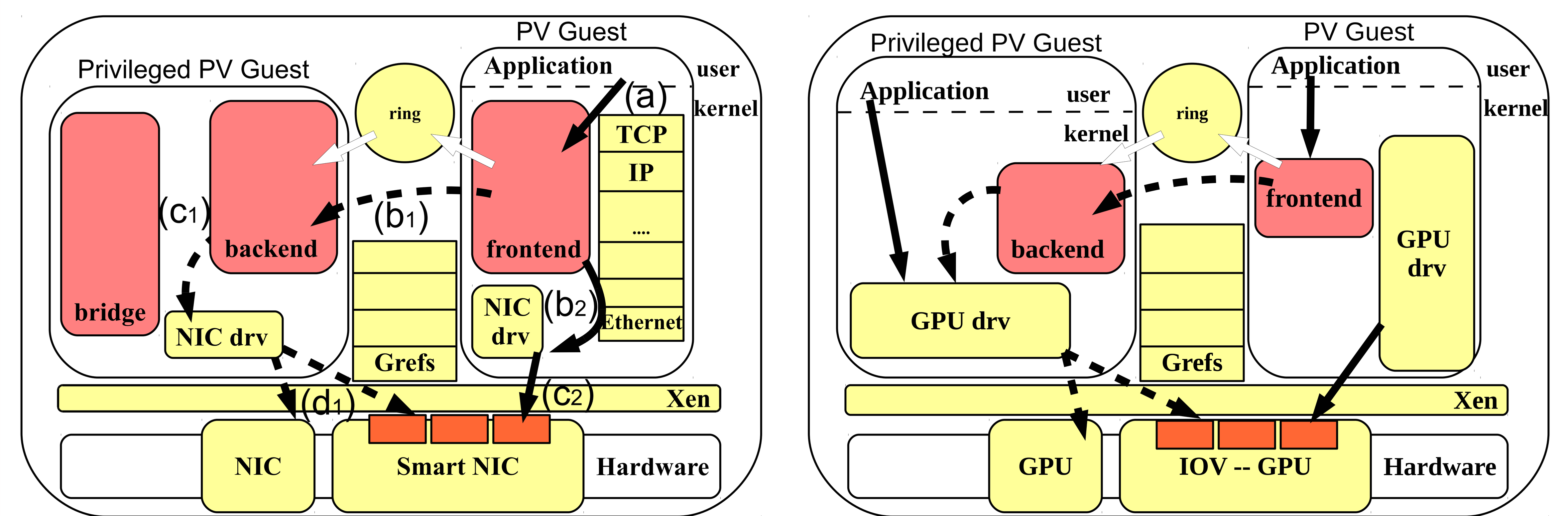
Hardware Offloading: Our prototype implementation consists of a frontend and a backend in the Xen virtualization platform. We show that while in the frontend part (VM) CPU time is negligible and agnostic to varying message sizes, there is still some CPU time spent in the Driver domain that can potentially reduce the performance of the entire system. This issue could be resolved by offloading this thin protocol layer to a hardware device, capable of performing DMA transfers and simple protocol processing.

Decoupling HPC from the Privileged Guest: While a hardware approach, as mentioned above, could resolve CPU time issues, we would like to examine the possibility of decoupling all HPC network-related issues to a different domain. This way, overheads associated with world switches and interrupt handling can be alleviated. We plan on evaluating this idea based on Xen stub domains, using modular networking components that can be easily implemented and optimized.

Acknowledgments

The authors would like to thank Nikos Nikoleris, Elisavet Kozyri and Stratos Psomadakis for their useful contributions to this work.

I/O Path

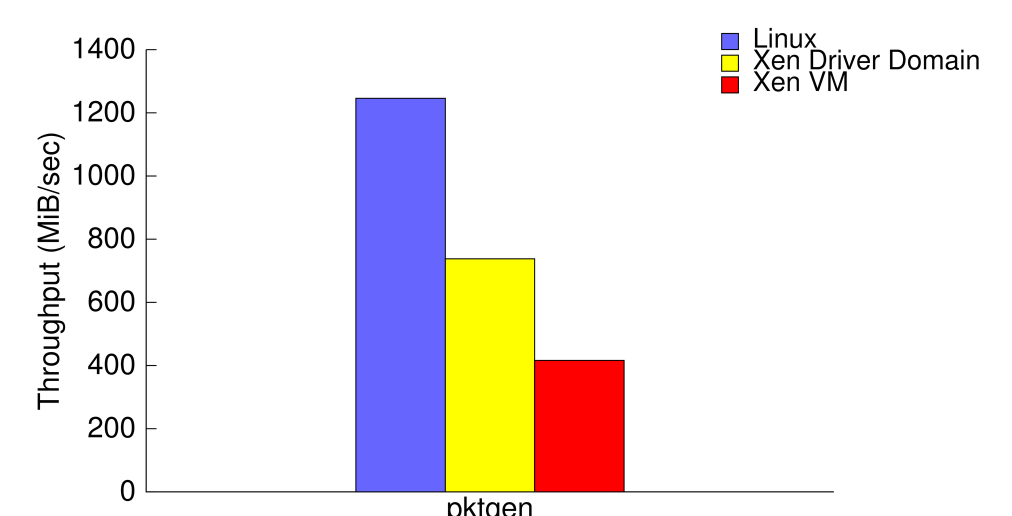


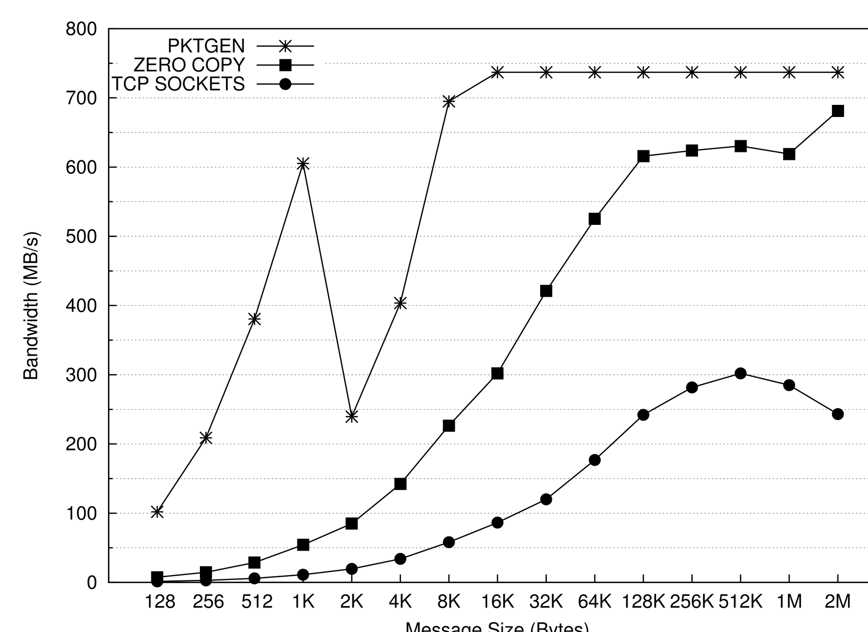
Applications issue access requests to the virtual device (frontend). The frontend passes requests to the backend using the event channel mechanism (dashed arrow, b_1). The backend performs the necessary operation, either registering memory buffers (filling up the IOTLB) or issues requests to native driver (dashed arrow, c_1). The driver, then, informs the device to DMA data from application to the on-chip buffers (dashed arrow, d_1).

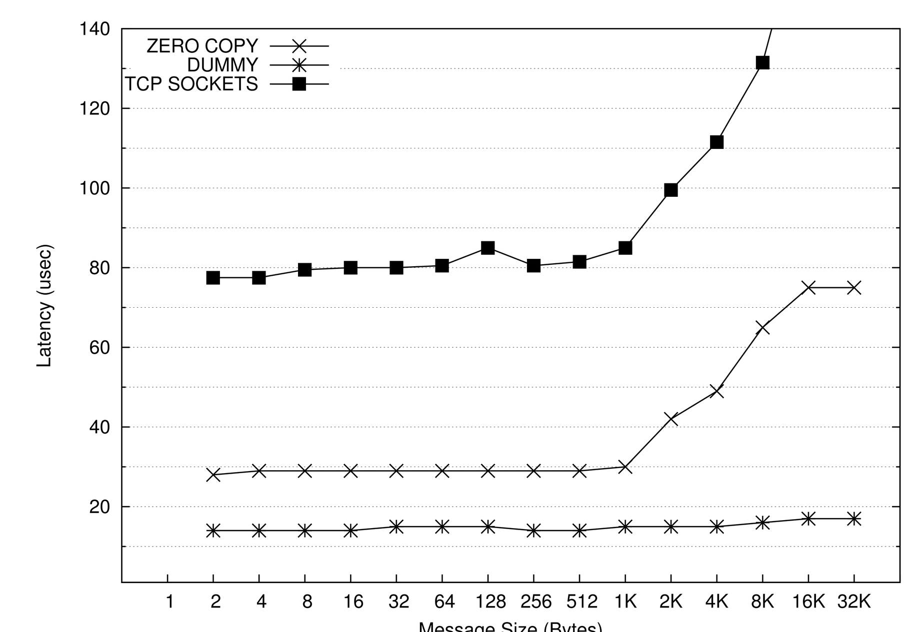
Although the proposed approach relaxes the system from processing and context-switch overheads, ideally, VMs could communicate directly with the hardware, lowering the multiplexing authority to the device's firmware. This could be realized using a smart controller that exports multiple queues or functions to the virtualization subsystem.

Preliminary results on HPC cluster interconnects

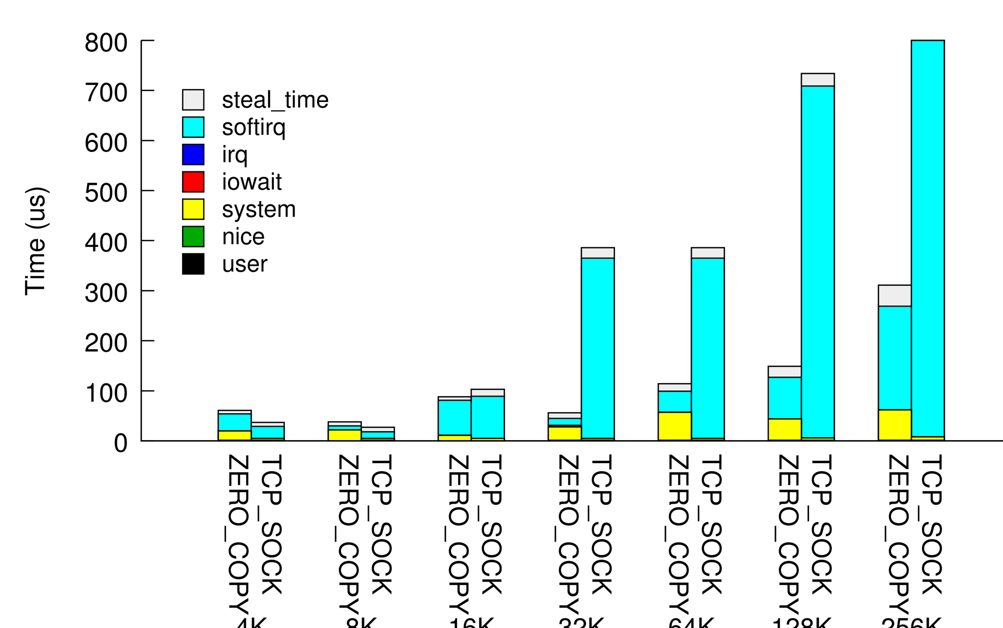
We run the pktgen utility of the Linux kernel. The figure on the right plots the maximum achievable socket buffer production rate when ran in vanilla Linux, inside the Privileged Guest and in the VM. In the default configuration, Xen's virtual ethernet device does not feature specific optimizations and as a result, its performance is limited to 416MiB/sec.

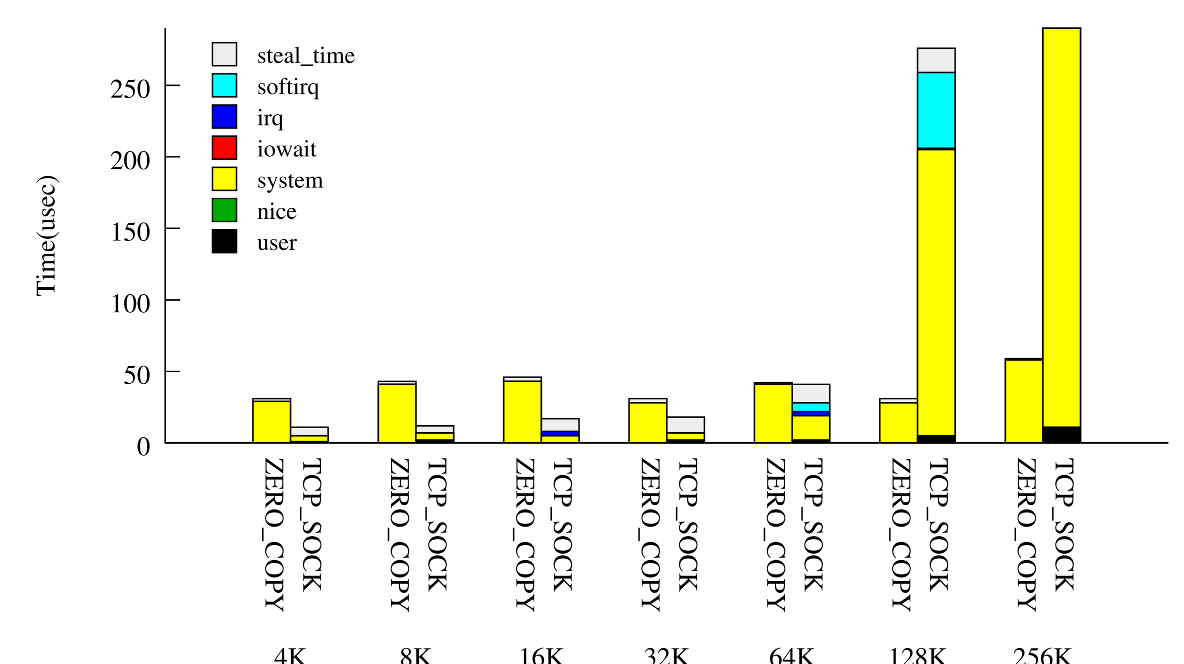






We use a custom synthetic microbenchmark to evaluate our approach over our interconnect sending unidirectional RDMA write requests. To obtain a baseline measurement, we implement our microbenchmark using TCP sockets. The figure on the left plots the aggregate throughput of the system. Specifically, we deployed our microbenchmark on top of TCP sockets (filled circles) and our framework (filled squares). An RDMA message over TCP sockets takes 77μsec to cross the network whereas over our framework takes 28μsec (Figure on the right).





We measure the total CPU time when two VMs perform RDMA writes of varying message sizes over the network (TCP and ZERO_COPY approach). To investigate the sources of CPU time consumption, we perform a detailed breakdown of the subsystems used in the RDMA operations for TCP and our framework. The figures above plot the CPU time breakdown for the Privileged Guest (left) and the VM (right).