Getting 10 Gb/s from Xen: Safe and Fast Device Access from Unprivileged Domains

Kieran Mansley, Greg Law, David Riddoch,
Guido Barzini, Neil Turton, and Steven Pope

Solarflare Communications, Inc.
{kmansley,glaw,driddoch,gbarzini,nturton,spope}@solarflare.com

Abstract. The networking performance available to Virtual Machines (VMs) can be low due to the inefficiencies of transferring network packets between the host domain and guests. This can limit the application-level performance of VMs on a 10 Gb/s network. To improve network performance, we have created a "virtualization aware" smart network adapter and modified Xen[1] to allow direct, but safe, access to such adapters from guest operating systems. Networking overheads are reduced considerably, and the host domain is removed as a bottleneck, resulting in significantly improved performance.
We describe our modifications to the Xen networking architecture that allow guest kernels direct — but secure — access to the networking hardware, whilst preserving support for migration. We also describe briefly how the same technology is used to grant direct network access to user-level applications and thus provide even greater efficiency in terms of bandwidth, latency and CPU utilisation.

1 Introduction

Modern commodity servers are able to saturate 10 Gb/s Ethernet networks [1]. However, VM guests incur significant additional overheads due to context-switching, data movement and passing packets between the host domain and guests. The Xen [2] virtual machine architecture imposes relatively low overheads on networking compared with other virtualization technologies, yet its performance is well below that of native operating systems [3, 4]. The privileged host VM quickly becomes a bottleneck as all network traffic must pass through it.
This I/O bottleneck makes virtualization effectively impractical for certain classes of application, including many HPC applications and high performance servers. As 10 Gb/s Ethernet moves more into the mainstream, the set of applications for which this is a barrier to adoption of virtualization will likely increase.
This paper shows how allowing guest operating systems direct access to the I/O hardware eliminates the bottleneck and software overheads usually associated with virtualized I/O, and allows a virtualized guest to achieve performance comparable with a system running natively. Doing

[1] Xen is a trademark of XenSource, Inc. in the United States and other countries.

so requires additional support from the hardware in order to multiplex the device between multiple guests that may access it concurrently and also to enforce isolation so that guests cannot gain privileges or compromise system integrity.
The rest of this paper is structured as follows. Section 2 outlines the architecture for accelerated networking that we have added to Xen, while Section 3 expands to describe our implementation and the results we have obtained with a virtualization-aware network adapter. Section 4 shows how the same techniques can be applied to user-level applications. Finally, Section 5 concludes.

2 Architecture

2.1 Xen Paravirtualized Network I/O

Paravirtualized network I/O in Xen is achieved through a pair of interlinked drivers; *netfront* the "frontend driver" in the guest, and *netback* the "backend driver" in the host domain. The frontend and backend communicate through a region of shared memory and send each other virtual interrupts using event channels. Together these form a channel that supports the transfer of packets between host domain and guest.
The upper edge of the frontend driver presents the interface of a standard network device driver, allowing it to interface to the bottom of the guest's network stack. The backend appears likewise and is usually configured to connect to a software bridge in the host OS. This allows it to communicate with the host's network stack, other virtual machines' backend drivers, and physical network interfaces and so the network beyond.
Packets that arrive from a physical network interface are routed by the bridge to the appropriate backend drivers, which in turn forward them to the corresponding guests' frontend drivers. These then pass them on to the network stack as if they had arrived directly at the guests. Packets sent by guests follow the same path in reverse. Packets sent from one guest to another are routed between the corresponding backend drivers by the bridge.

2.2 Acceleration Architecture

We have extended the Xen netfront/netback architecture with a plugin interface that provides an opportunity to accelerate network performance. We have preserved the existing netfront/netback channel and added an optional "fast path" implemented by the plugins. To account for as many different kinds of hardware as possible (existing and future designs), the plugin interface makes as few assumptions as possible. By preserving the existing netfront/netback mechanism it is possible to support migration and also hardware that accelerates only a subset of traffic with other traffic taking the "slow path".
Various acceleration techniques are possible, but the main model anticipated is for the plugin driver in the guest to access the network adapter

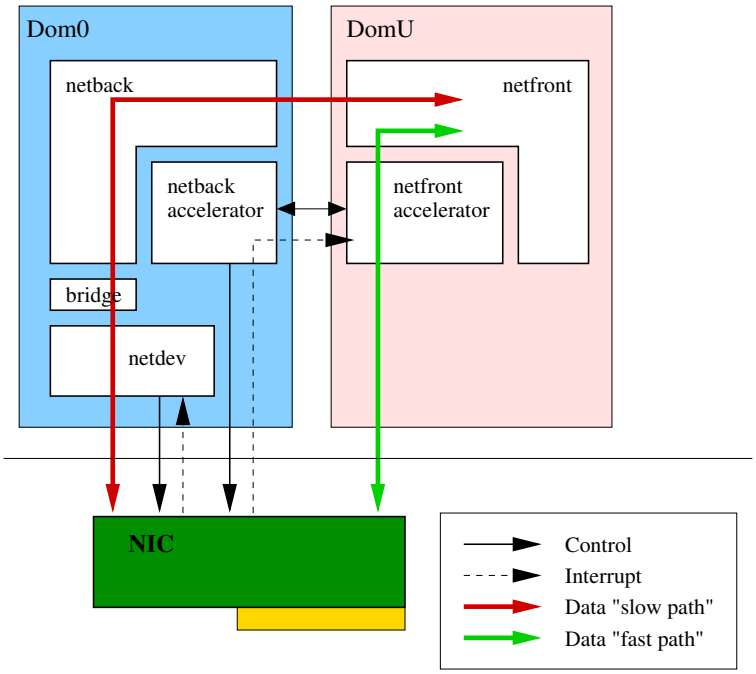

Fig. 1. Accelerated networking in Xen

hardware directly to send and receive packets, bypassing the host domain and associated overheads. This is illustrated in Figure 1.

The netfront and netback drivers each accept an "accelerator plugin." The frontend accelerator communicates with netfront and implements the data path, whereas the backend accelerator communicates with netback and handles control functions.[2] See Section 3 for details of our implementation of accelerators for the Solarstorm SFC4000 controller.

Accelerated Transmit Path. Packets to be transmitted are passed from the guest kernel to the netfront driver. If an accelerated plugin is present it is given the opportunity to send the packet via the fast path, and it indicates whether or not it did so. If it did not, the netfront driver transmits the packet via the slow path through netback. Thus netfront need have no knowledge of the capabilities of the accelerator. Packets that are destined for local VMs (those in the same system) must not be transmitted onto the network and so in most cases are sent via the slow path. However, some smart adapters might be able to deliver them via an accelerated path.

[2] The backend accelerator can be part of the network adapter's 'net' driver, or can be a separate module.

Accelerated Receive Path. How the receive path is accelerated depends on the capabilities of the hardware and frontend accelerator. Some smart network adapters are able to demultiplex received packets to the appropriate guest by inspecting headers and delivering packets directly into the guest's address space. The frontend accelerator may then be invoked by an interrupt or via the host domain and an event channel. It then passes received packets to the kernel stack. The plugin interface allows the frontend accelerator to participate in Linux's NAPI algorithm for managing interrupts.

If the network adapter is not able to deliver a packet directly to a guest it can deliver it to the net driver in the host domain, from where it will be passed along the normal path through the bridge and netback drivers. This path will usually be used for multicast and broadcast packets that need to be delivered to multiple guests. When a received packet is delivered via netback, it is presented to the backend accelerator. This gives the accelerator an opportunity to inspect the packet and if appropriate program the network adapter to deliver similar packets directly to the guest in future. This provides a means to support hardware with differing means of demultiplexing received traffic and also for the backend accelerator to allocate scarce hardware resources according to demand.

By placing these small hooks in the netfront/netback architecture, we aim to leave sufficient flexibility for device vendors to construct a wide range of acceleration devices and drivers. That is, our goal is to make the netfront/netback drivers in Xen simple and device-agnostic. This keeps the acceleration-specific and device-specific code in the accelerator plugins and thus achieves maximum flexibility.[3]

The code that implements the framework for this architecture was recently accepted by XenSource into the xen-unstable open source repository and will likely be included in future releases of Xen.

2.3 Security

The security concerns that this architecture raises are largely to do with the guest OS's ability to access network hardware directly. It must not be possible for software running in the guest VM, whether by accident or attack, to compromise the security, integrity and isolation guarantees normally provided by Xen. To prevent this, the network adapter must be able to provide the following guarantees:

1. Guests can only transfer network data to and from the adapter using memory regions they own. This may be enforced by the use of a platform IOMMU, or by pre-registering buffers through the host domain (which can do the necessary checks) and restricting access to that set.
2. Packets must only be delivered via a fast path if they would have reached the same recipients via the slow path. I.e. packets must only be delivered to their addressee.

[3] Once several accelerator plugin drivers have been developed, it may be that common code is identified and a framework is built to encapsulate this common code. However, that is beyond the current scope of this work.

3. Guests can only transmit packets which have the correct source MAC address.

Other features may also be desirable, including filtering packets according to other address fields such as VLAN tag or IP address. The ability to control the rate at which individual guests are able to transmit can also be useful to achieve fair access to the network amongst competing VMs.

2.4 Migration

Exposing entire devices directly to the guest (by "PCI pass-through" for example) would render migration between machines with heterogeneous hardware very difficult. Fortunately, the use of the para-virtualized netfront/netback driver model makes migration relatively straightforward.

The plugin model and ability to fall back to the slow path are key. Before a VM is moved from one host to another the frontend accelerator plugin is removed and all network traffic reverts to the slow path. The VM no longer has direct access to the hardware and so can be migrated without difficulty. Note that the removal of the frontend accelerator is largely transparent to applications in the guest VM, which at worst see a degradation in network performance.

On arrival at the new host, the netfront/netback channel is created and networking on the slow path resumes. If the new host contains a smart adapter a suitable frontend accelerator is loaded once again and networking can be accelerated.

The acceleration architecture must also be robust as other VMs migrate out-of and into a host that contains an accelerated VM. When another VM migrates in, the existing frontend accelerators need to be aware of this transition as they must switch from accepting packets on the fast path to sending them down the slow path so they can be delivered locally.[4] Without this it would continue to route packets via the fast path onto the network when the destination is now on the same host.

This is the same problem as that facing Ethernet switches in the network, as they must also become aware of the change in topology. The solution takes advantage of the gratuitous ARP packet sent after migration to update the switches. On receipt of this ARP the backend accelerator plugin can inform the frontend plugin of the change.

3 Implementation for Solarstorm Controllers

Presently the Solarstorm SFC4000 is the only device for which accelerator plugins have been written, but it is likely that other adapters will be supported in the future, particularly with the advent of platform IOMMUs [5, 6] and PCI-IOV [7] devices.

[4] Unless the smart adapter itself supports local deliver in hardware.

3.1 Hardware

Like most modern network adapters, the SFC4000 uses DMA descriptor rings to manage transfer of packet data between host memory and the adapter. It has another type of queue, called an *event queue*, to notify the software as DMA transfers complete. In common with some other high-end adapters, the SFC4000 supports many DMA queues and event queues. A transmit ring, receive ring and event queue together form a virtual interface (VI) which comprises all of the resources needed to transfer packets between the host and network. Each VI is accessed through a separate page of the I/O address mapping of the adapter and so an unprivileged domain can be given access to just one VI by mapping the I/O page into the domain's address space.

The SFC4000 steers received packets to the appropriate VI's receive queue by inspecting address fields in packet headers, under the control of *filters*. Filters can only be programmed by the backend accelerator in the priviliged host domain.

In contrast to most other network adapters, the addresses of DMA buffers in host memory can be specified either with physical addresses, or protected virtual addresses. The virtual addressing mode is used for VIs that are mapped into unprivileged domains and provides the memory protection guarantee discussed in Section 2.3. An internal IOMMU is used to translate the virtual addresses to physical addresses, which can only be programmed by the privileged driver.

3.2 Accelerated Transmit

Accelerated transmit is relatively straightforward. Most packets offered by netfront to the accelerator plugin are simply appended to the transmit ring of its VI. However, the frontend plugin elects not to transmit packets addressed to other guests on the same host, including broadcast and multicast packets (netfront will then send them in the normal way). Once the DMA transfer for a send is completed, the adapter places a completion notification on the event queue, which the frontend accelerator responds to by releasing the packet buffer.

3.3 Accelerated Receive

Each packet received by the adapter is directed to a DMA queue according to its addressing information. If the packet's destination address matches a filter, the packet is received onto the VI with which that filter is associated. If the destination address matches no filters, the packet is received onto the default VI.

Packets received onto the default VI queue are handled by the net driver. These packets are passed to the backend accelerator plugin in order to give it an opportunity to accelerate future traffic with that addressing information. The backend accelerator will typically insert a filter to ensure that future packets received to this address are steered by the hardware

to the appropriate VI for the guest. The net driver then passes the received packet up to the network stack as usual, which will then make its way over netback to the appropriate netfront in the traditional Xen way. Currently Xen does not support MSI-X, so all interrupts are delivered to the net driver, even for accelerated traffic. The net driver must then notify the appropriate frontend accelerator (via the backend accelerator) using an event channel.[5] Upon receiving the event channel interrupt, the guest's frontend accelerator inspects its event queue and delivers received packets to the guest OS.

3.4 Performance

We used the Chariot [8] benchmark suite to measure the TCP bandwidth achievable with 1–4 guests, and compared the SFC4000 accelerator plugin implementation against the same (but unmodified) version of the latest open source xen-unstable tree. The experiments were run on two Dell PowerEdge 2950 servers, each fitted with two quad-core Intel Xeon 2.66 GHz CPUs. The CPUs were partitioned such that each VM was pinned to its own (exclusive) physical CPU core and each virtual machine had 256 MB of memory. The servers were connected back-to-back using a pair of Solarstorm SFE4003 CX-4 10 Gb/s Ethernet adapters. The operating systems used were based on Red Hat Fedora Core 5 modified to use the open source xen-unstable tree with Linux 2.6.18-xen kernels. The MTU was 1500 bytes. For tests involving multiple guests, each guest was running the Chariot benchmark client concurrently. All throughput results are the aggregate for all guests.
Table 1 shows results for a single uni-directional TCP stream to each guest. Performance saturates at 9.25 Gb/s due to a PCIe bus bandwidth limitation. Table 2 shows corresponding results for bi-directional traffic.

Table 1. Bandwidth (Gb/s) for uni-directional traffic.

Number of guests	Unaccelerated throughput Gb/s	Accelerated throughput Gb/s
1	1.93	5.26
2	2.60	8.86
3	2.80	9.25
4	2.84	9.25

We also noted that under heavy load with unmodified xen-unstable the host domain became very unresponsive to the point of being very difficult to use. This is thought to be due to it becoming overloaded by the

[5] We plan to implement MSI-X support in the near future, meaning that the hardware will be able to deliver interrupts directly to the guest.

Table 2. Bandwidth (Gb/s) for bi-directional traffic.

Number of guests	Unaccelerated throughput Gb/s	Accelerated throughput Gb/s
1	2.13	5.34
2	2.66	9.95
3	2.85	11.61
4	2.91	14.22

network traffic passing through it. In the accelerated experiments there was no human-observable effect on the host domain's responsiveness as the network load was carried by the guests' CPUs.

This illustrates a further benefit of accelerated virtualized networking: Each guest has an independent network stack and so network traffic passed to and from the physical network is multiplexed only in the hardware. Thus there is no cross-talk between CPUs that are running separate guests and performance scales very well as the number of CPU cores and guests increases.

4 Solarstorm and User-Level Networking

The same hardware features that make the Solarstorm adapters suitable for direct access by untrusted guest VMs can be used to allow direct access by untrusted user-level applications. We have developed a TCP stack which can be linked against user-level applications and allows the bulk of network processing to be performed in the context of the application, eliminating system calls from the transmit and receive data paths. This results in significant reduction in CPU utilisation, increased bandwidth and reduced latency. We call our user-level TCP/IP stack *Open Onload*, and intend to release it under an open-source licence in the near future.

The plot in Figure 2 shows the bandwidth achieved by Open Onload compared with the Linux kernel stack running over the same hardware. This test was run with a 1500byte MTU and reaches link saturation bandwidth.

The application to application one-way latency for the kernel stack[6] on these systems is 10.1 μs and just 5.3 μs with Open Onload. Of this, 4.3 μs is attributable to the hardware and link, so the software overhead to send and receive a small message is 5.8 μs for the kernel stack and just 1.0 μs for Open Onload.

As well as improving performance, the Open Onload stack improves system behaviour by performing network processing in the context of the application. This ensures that the work done is properly accounted to the

[6] With interrupt moderation disabled.

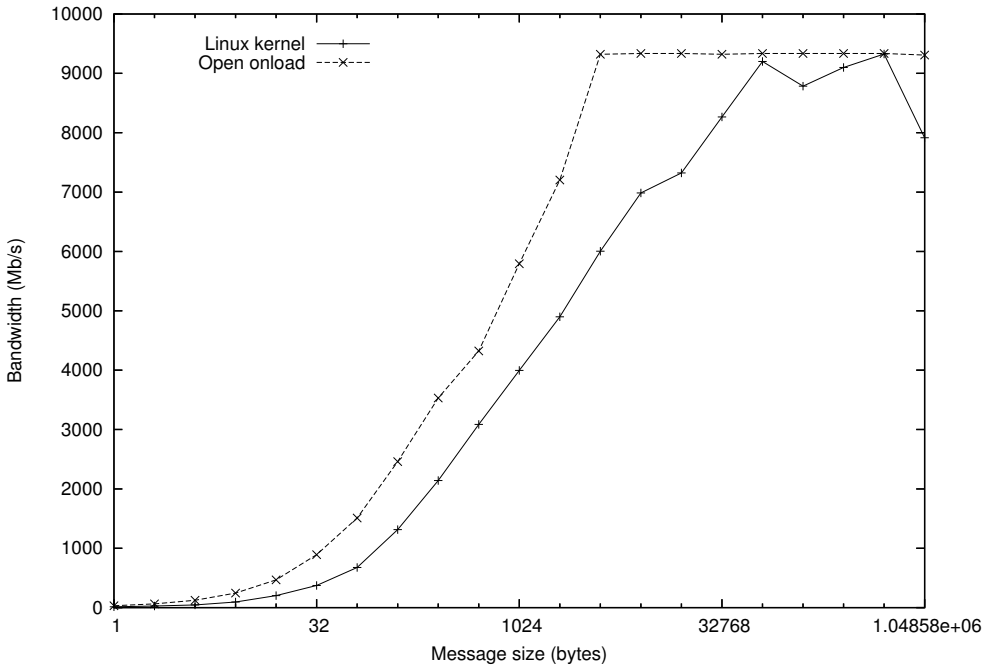

Fig. 2. Bandwidth (Mb/s) of user-level and kernel networking

application and so improves fairness. Another consequence is that it does not suffer from receive live-lock [9], a situation in which a CPU spends all of its time processing network traffic at high priority and not making any progress in the user-level application. Thus performance under overload conditions is improved.

5 Conclusion

This paper has presented an architecture for accelerating network access from virtual machines. The architecture has several advantages:

Fast The performance delivered with the acceleration architecture far exceeds that seen by "vanilla" Xen, particularly when many guests are competing for network access. By removing the shared host domain from the common data path, performance isolation is improved.

Simple The amount of additional code in the Xen mainline has been kept low.

Flexible The architecture makes minimal assumptions about the capabilities offered by hardware, meaning it should be compatible with a wide range of devices, present and future.

Mobile The ability to fall-back to the slow path means it is possible to support migration in heterogeneous environments and benefit from acceleration when suitable hardware is available.

Safe With suitable hardware support the architecture ensures that VMs remain isolated from each other and can not exploit their direct access to the NIC to interfere with other guests.

Using this architecture Xen can now fill a 10 Gb/s pipe and deliver performance to guest OSs comparable to that which is normally only seen by the host OS. Further developments currently in progress, including MSI-X support, are expected to improve performance yet further.

References

1. Steven Pope and David Riddoch: 10Gb/s Ethernet performance and retrospective. Computer Communication Review **37**(2) (2007)
2. Pratt, I., Fraser, K., Hand, S., Limpach, C., Warfield, A., Magenheimer, D., Nakajima, J., Mallick., A.: Xen 3.0 and the Art of Virtualization. Proceedings of the Ottawa Linux Sumposium (2005)
3. Menon, A., Cox, A., Zwaenepoel, W.: Optimizing Network Virtualization in Xen. In: USENIX Annual Technical Conference. (2006)
4. Menon, A., Santos, J.R., Turner, Y., Janakiraman, G.J., Zwaenepoel, W.: Diagnosing Performance Overheads in the Xen Virtual Machine Environment. In: ACM/USENIX Conference on Virtual Execution Environments (VEE05). (june 2005)
5. Intel Corportation: Intel Virtualization Technology for Directed I/O. `http://www.intel.com/technology/itj/2006/v10i3/2-io/5-platform-hardware-support.htm` (2006)
6. AMD Inc: AMD I/O Virtualization Technology (IOMMU) Specification. `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/34434.pdf`
7. PCI SIG: PCI-SIG - I/O Virtualization. `http://www.pcisig.com/specifications/iov/`
8. Ixia Corporation: IxChariot. `http://www.ixiacom.com/products/display.php?skey=ixchariot`
9. Mogul, J.C., Ramakrishnan, K.: Eliminating receive livelock in an interrupt-driven kernel. ACM Transactions on Computer Systems **15**(3) (1997) 217–252