High Performance and Scalable I/O Virtualization via Self-Virtualized Devices

Himanshu Raj and Karsten Schwan
CERCS, College of Computing, Georgia Tech
Atlanta, GA, USA
{rhim,schwan}@cc.gatech.edu

ABSTRACT

While industry is making rapid advances in system virtualization, for server consolidation and for improving system maintenance and management, it has not yet become clear how virtualization can contribute to the performance of high end systems. In this context, this paper addresses a key issue in system virtualization – how to efficiently virtualize I/O subsystems and peripheral devices. We have developed a novel approach to I/O virtualization, termed *self-virtualized devices*, which improves I/O performance by offloading select virtualization functionality onto the device. This permits guest virtual machines to more efficiently (i.e., with less overhead and reduced latency) interact with the virtualized device. The concrete instance of such a device developed and evaluated in this paper is a self-virtualized network interface (SV-NIC), targeting the high end NICs used in the high performance domain. The SV-NIC (1) provides virtual interfaces (VIFs) to guest virtual machines for an underlying physical device, the network interface, (2) manages the way in which the device's physical resources are used by guest operating systems, and (3) provides high performance, low overhead network access to guest domains. Experimental results are attained in a prototyping environment using an IXP2400-based ethernet board as a programmable network device. The SV-NIC scales to large numbers of VIFs and guests, and offers VIFs with ∼77% higher throughput and ∼53% less latency compared to the current standard virtualized device implementations on hypervisor-based platforms.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems—*Design studies, Performance attributes*; D.4.4 [**Software**]: Operating SystemsCommunications Management[Network communication]

HPDC'07, June 25–29, 2007, Monterey, California, USA.

General Terms

Measurement, Performance, Design, Experimentation, Management

Keywords

Virtualization, Virtual Devices

1. INTRODUCTION

Virtualization technologies have long been used for high end server class systems, examples including IBM's pSeries and zSeries machines. More recently, these technologies are becoming an integral element of processor architectures for both lower end PowerPCs and x86-based machines [6]. In all such cases, the basis for machine virtualization are the hypervisors or Virtual Machine Monitors (VMMs) that support the creation and execution of multiple guest virtual machines (VMs or domains) on the same platform and enforce the isolation properties necessary to make the underlying shared platform resources appear exclusive to each domain. Examples of these are Xen [18] and VMWare ESX server [11]. The virtualization methods used include resource partitioning, time sharing, or a combination thereof, and they are used to (i) create virtual instances of all physical resources, including the peripheral devices attached to the host, and (ii) dynamically manage virtualized components among the multiple guest domains.

The purpose of our research is to explore the implications of virtualization for the high performance domain, with our initial focus being the virtualization of peripheral resources. Current implementations of I/O device virtualization use a driver domain per device [32], a driver domain per set of devices [27], or they run driver code as part of hypervisor itself [18]. The latter approach has been dismissed in order to avoid hypervisor complexity and the increased probability of hypervisor failures caused by potentially faulty device drivers.

It is clear that current approaches to device virtualization differ from the way high performance or petascale operating systems and applications operate, where devices are directly mapped into user space [26] and applications have full control over device operation. Instead, with current virtualized systems, the unfortunate fact for HPC is that their use of 'driver domain'-based I/O requires multiple domains to be scheduled for each device access, with consequent negative implications on overhead and cache behavior [25]. Youseff et al. demonstrate substantial performance degradation for MPI benchmarks for systems virtualized with Xen [39].

This paper introduces an alternative, high performance solution for I/O virtualization, termed self-virtualized devices. Using the processing elements present on the higher end devices commonly present in the HPC domain, *a self-virtualized device implements selected virtualization functionality on the device itself*, resulting in reduced host involvement and overheads and improved latency in device access. As a concrete example of a self-virtualized device, we have created a self-virtualized network interface (SV-NIC) on an implementation platform comprised of a SMP machine using a high end communication device. To evaluate alternative offloading solutions, the programmable communication device chosen is an IXP2400 network processor-based gigabit ethernet board. The IXP2400 network processor contains multiple internal processing elements situated *close* to the physical I/O device, which the SV-NIC exploits to implement virtualization solutions imposing low overheads on hosts and offering levels of performance exceeding that of the I/O virtualization solutions based on the 'driver-domain' approach. The implementation also frees host computational resources from simple communication tasks, thereby permitting them to run the computationally intensive application codes for which they are designed. For server systems, self-virtualization improves platform scalability, permitting a large number of guest virtual machines to efficiently share the platform's I/O devices.

The scalability and performance of self-virtualized devices depend on two key factors: (1) the virtual interface (VIF) abstraction and its associated API, and (2) the algorithms used to manage multiple virtual devices. For the SV-NIC, measured performance results show it to be highly scalable in terms of resource requirements. Specifically, limits on scalability are not due to the design, but are dictated by the availability of resources, such as the physical communication bandwidth and the maximum number of processing cores that can be deployed by the device.

While initially focused on SV-NICs, our research has already gone beyond virtualizing high end network devices, by generalizing the notion of SV-NIC to that of a self-virtualized I/O abstraction (SV-IO). This abstraction captures all of the functionality involved in virtualizing an arbitrary peripheral device, the intent being to make it easy to flexibly map such functionality to the multiple processing cores present in today's multi-core or tomorrow's many-core platforms. In this fashion, SV-IO functions can be executed on general purpose cores, on specialized cores, or on the high end devices explored in the experiments presented in this paper. In all such cases, to guest operating systems, this functionality and its implementation are 'hidden' behind a simple, yet flexible virtual device interface abstraction.

In summary, this paper makes the following contributions:

- It introduces the self-virtualized device approach for high-performance I/O virtualization;
- it presents a realization of a self-virtualized device for a high end network device. By using this device's internal processing resources, concurrency in the SV-NIC implementation results in high performance and scalability for the virtual interfaces (VIFs) used by guest operating systems and their applications; and
- it extends the notion of self-virtualized device to that of self-virtualized I/O, to allow flexible, efficient mappings of virtualization functionality to future many-core systems.

Performance results demonstrate that the SV-NIC permits virtual interfaces to operate at full link speeds, within limits dictated by the PCI bandwidths attainable with our current prototype configuration. At gigabit link speeds, for instance, current PCI performance dominates, so that a VIF from our SV-NIC provides TCP throughput and latency of $\sim$ 620Mbps and $\sim$ .076ms, respectively. Interestingly, this is still $\sim$77% higher throughput and $\sim$53% lower latency compared to a virtual device implemented with the 'driver domain' approach. Performance scaling is excellent, permitting the SV-NIC to easily deal with an increasing number of virtual interfaces and guest domains using them. For example, for 8 VIFs, the aggregate throughput for the SV-NIC is 103% higher compared to the 'driver-domain'-based approach, while the average latency is 39% less.

2. SELF-VIRTUALIZED I/O DEVICES

A self-virtualized I/O device is a peripheral with additional computational resources *close* to the physical device. The device utilizes these resources to encapsulate certain I/O virtualization functionality. In particular, the device is responsible for:

- scalable multiplexing/demultiplexing of a large number of *virtual devices* mapped to a single physical device;
- providing a lightweight API to the hypervisor for managing virtual devices;
- efficiently interacting with guest domains via simple APIs for accessing the virtual devices; and
- harnessing the compute power (i.e., potentially many processing cores) offered by the underlying hardware platform.

Before we describe the different components of a self-virtualized device and their functionalities, we briefly digress to discuss the virtual interface (VIF) abstraction provided by the device and the associated API for accessing a VIF from a guest domain.

2.1 Virtual Interfaces (VIFs)

Examples of *virtual I/O devices* on virtualized platforms include virtual network interfaces, virtual block devices (disk), virtual camera devices, and others. Each such device is represented by a *virtual interface* (VIF) which exports a well-defined interface to the guest OS, such as ethernet or SCSI. The virtual interface is accessed from the guest OS via a VIF device driver.

Each VIF is assigned a *unique ID*, and it consists of two message queues, one for outgoing messages to the device (i.e., *send queue*), the other for incoming messages from the device (i.e., *receive queue*). The simple API associated with these queues is as follows:

```
boolean isfull(send queue);
size_t send(send queue, message m);
boolean isempty(receive queue);
message recv(receive queue);
```

The functionality of this API is self-explanatory.

A pair of signals is associated with each queue. For the send queue, one signal is intended for use by the guest domain, to notify the self-virtualized device that the guest has enqueued a message in the send queue. The other signal is used by the device to notify the guest domain that it has received the message. The receive queue has signals similar

to those of the send queue, except that the roles of guest domain and self-virtualized device are interchanged. A particular self-virtualized device need not use all of these defined signals. For example, if the device polls the send queue to check the availability of messages from the guest domain, it is not required to send the signal from guest domain to the device. Furthermore, queue signals are configurable at runtime, so that they are only sent when expected/desired from the other end.

2.2 Design

A self-virtualized device has three components – the *processing component*, that consists of one or more cores, and connects to the *physical I/O device* via the *peripheral communication fabric*. The device itself is connected to the host system via a *messaging fabric*, which guest domains utilize to communicate with the device using VIFs.

The two main functions of a self-virtualized device are *managing VIFs and performing I/O*. Management involves creating or destroying VIFs or reconfiguring various parameters associated with them. These parameters define VIF performance characteristics, and in addition, they can be used by guest domains to specify QoS requirements for the virtual device.

When performing I/O, in one direction, a message sent by a guest domain over a VIF's send queue is received by the device's processing component. The processing component then processes the message and forwards it to the physical device over the peripheral communication fabric. Similarly, in the other direction, the physical device sends data to the processing component over the peripheral communication fabric, which then demultiplexes it to one of the existing VIFs and sends it to the appropriate guest domain via the VIF's receive queue.

A key task in processing message queues is for the self-virtualized device to multiplex/demultiplex multiple VIFs on a single physical I/O device. The scheduling decisions made as part of this task must enforce performance isolation among different VIFs. While there are many efficient methods for making such decisions, e.g. the DWCS [35] algorithm developed in our own past research, the simple scheduling method used in this paper's experimentation is round-robin scheduling. A detailed study of VIFs' performance and fairness characteristics with different scheduling methods is beyond the scope of this paper.

3. THE SELF-VIRTUALIZED NETWORK INTERFACE (SV-NIC)

In this section, we present a concrete self-virtualized device implementation for a high end network device. The device used is an IXP2400 network processor(NP)-based RadiSys ENP2611 board [1]. This resource-rich network processor features a XScale processing core and 8 RISC-based specialized communication cores, termed *micro-engines* (although not all of them are currently utilized.) Each micro-engine supports 8 hardware contexts with minimal context switching overhead. The XScale core and the micro-engines form the processing component of the SV-NIC. The physical network device on the board is a PM3386 gigabit ethernet MAC and is connected to the NP via the *media and switch fabric* (MSF) [12], which forms the peripheral communication fabric. The board also contains onboard memory, including

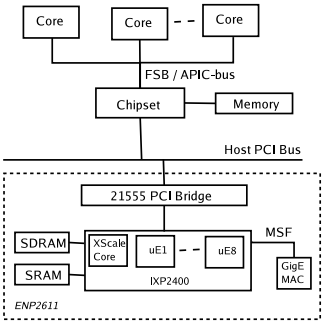

Figure 1: NP-based network interface and the host platform

SDRAM, SRAM, scratchpad and micro-engine local memory (listed in the order of decreasing sizes and latencies, and increasing costs.) The board runs an embedded Linux distribution on the XScale core, which contains, among others, some management utilities to execute *micro-code* on the micro-engines. This micro-code is the sole execution entity that runs on the micro-engines.

The board utilizes the host PCI bus as the messaging fabric, i.e., it is connected to the host PCI bus via the Intel 21555 non-transparent PCI bridge [5]. This bridge allows the NP to access only a portion of host RAM resources via a 64MB PCI address window. In contrast, host cores can access all of the NP's 256MB of DRAM. Figure 1 shows the block diagram of our combined ENP2611 and host platform.

The following details about the PCI bridge are relevant to some of our performance results. The PCI bridge contains multiple *mailbox* registers accessible from both host- and NP-ends. These can be used to send information between host cores and NP. The bridge also contains two interrupt identifier registers called *doorbell*, each 16-bit wide. The NP can send an interrupt to the host by setting any bit in the host-side doorbell register. Similarly, a host core can send an interrupt to the *XScale core* of the NP by setting any bit in the NP-side doorbell register. Although the IRQ asserted by setting bits in these registers is the same, the IRQ handler can differentiate among multiple "reasons" for sending the interrupt by looking at the bit that was set to assert the IRQ.

As stated earlier, guest domains access a self-virtualized device via its VIF abstraction. For the SV-NIC, the send queue of each VIF is used for outgoing packets from guest domains, and the receive queue is used for incoming packets. Due to asymmetric PCI performance in our platform, with PCI writes heavily favored over PCI reads [21], our current implementation avoids PCI reads whenever possible. In particular, the send queue is placed into the NP's SDRAM, while the receive queue is implemented in host memory. The VIF device driver for guest domains also has the capability to map these queues directly into the application's address space. This provides the ability to bypass the operating system's networking stack, thereby also enabling high performance communication from user space.

As explained in more detail in the next section, only some of the signals associated with VIFs are needed: those sent from the SV-NIC to the guest domain. These two signals work as transmit and receive interrupts, respectively, similar to what is needed for physical network devices. Both

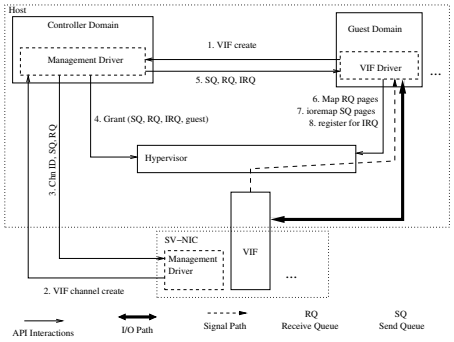

Figure 2: Management interactions between the SV-NIC, hypervisor and the guest domain to create a VIF.

signals are configurable and can be disabled/enabled at any time by the guest domain virtual interface driver, as required. For example, the send code of the guest domain driver does not enable the transmit interrupt signal till it finds that the send queue is full (which will happen if the SV-NIC is slower than the host processor). Similarly, the receive code of the guest domain driver uses the NAPI [31] interface and disables receive interrupt signal when processing a set of packets. This reduces the interrupt load on the host processor when the rate of incoming packets is high. The queues have configurable sizes that determine transmit and receive buffer lengths for the store and forward style communication between the SV-NIC and guest domain.

3.1 Functionality breakdown for SV-NIC's processing component

This section describes how the cores used for the processing component of the SV-NIC achieve (1) VIF management, and (2) network I/O.

Management functionality includes the creation of VIFs, their removal, and changing attributes and resources associated with them. Figure 2 depicts various *management interactions* between the SV-NIC's processing component, the hypervisor and the guest domain to *create a VIF*. The hypervisor enhances guest domain privileges and enforces security isolation, which is further discussed in Section 3.2. The figure also shows the I/O and signaling paths for the VIF between the SV-NIC and the guest domain (via the messaging fabric).

Other management functionality includes the *destruction of VIFs* and *changing attributes* of a VIF or of the SV-NIC. Destruction requests are initiated by the hypervisor when a VIF has to be removed from a guest. This might be the result of a guest VM shutdown, or for security reasons (e.g. when a VM is compromised, its VIFs can be torn apart.)

Certain *attributes* can be set, at VIF creation time or later, to change VIF properties. For example, the throughput achievable by a VIF directly depends on the *buffer space* provided for the send- and receive-queues. Bandwidth and latency also depend on the *scheduling algorithm* used at the NP for the processing of packets corresponding to different VIFs. Hence, changing these attributes will affect runtime changes in VIF behavior.

Management functionality is accomplished by two management drivers – one running on the host and the other running on the SV-NIC. The host-side driver runs as part of the controller domain (dom0). The device-side driver is part of the embedded OS running on the NP-based board. It runs on the XScale core.

Management requests are generated by guest domains or the hypervisor. They are forwarded to the host-side management driver, which in turn forwards relevant parameters to the device-side driver via the 21555 bridge's mailbox registers. The device-side driver appropriates the resources for VIFs, which includes assigning micro-engines for network I/O and messaging fabric space for send/receive queues. The device-side driver then communicates these changes to the host-side driver, via the bridge's mailbox registers, and to the micro-code running on the micro-engines, via SRAM.

A guest domain performs network I/O via a VIF. It enqueues packets on the VIF's send-queue and dequeues packets from the VIF's receive-queue. It is the responsibility of the SV-NIC to:

- *egress:* multiplex packets in the send-queues of all VIFs on to the network device; and
- *ingress:* demultiplex the packets received from the network device onto appropriate VIFs' receive queues.

Since VIFs export a regular ethernet device abstraction to the host, this implementation models a software layer-2 (ethernet) switch.

Our current implementation uses four micro-engines for network I/O – one for egress, one for ingress and two for executing the device driver to perform I/O on the gigabit NIC.

Egress is managed by one micro-engine context per VIF. For simple load balancing, this context is selected from a pool of contexts belonging to a single micro-engine (the egress micro-engine) in a round robin fashion. Hence, the lists of VIFs being serviced by the contexts of the egress micro-engine are mutually disjoint. This allows for lock free operation of all contexts. The context dequeues the packet from the send-queue of the VIF and forwards it to the device driver. Contexts managing egress employ *voluntary* yielding after processing every packet and during packet processing for I/O to maintain a fair-share of physical network resources across multiple VIFs.

Ingress is managed for all VIFs by a shared pool of contexts belonging to one micro-engine (the ingress micro-engine). Each context selects a packet from the physical network, *demultiplexes* it to a VIF based on MAC address, locks the VIF, obtains a *sequence number* to perform "in-order" placement of packets, unlocks the VIF, and signals the *next* context to run. Next it performs the I/O action of moving the packet to the VIF receive-queue, during which it voluntarily relinquishes the micro-engine to other contexts that are either performing I/O or waiting to get a chance to execute. After a context is done performing I/O, it waits for the expected sequence number of the VIF to match its sequence number, yielding the micro-engine voluntarily between checking for this condition to become true. Once this wait operation is complete, the context atomically adjusts the VIF's receive-queue data structures to signify that a packet is successfully queued. Also, a signal to the guest domain is sent if required by the guest domain driver.

The SV-NIC sends signals to the guest domain, and its micro-engines poll for information from the guest domains. There are multiple reasons for this design: (1) an ample number of micro-engines and fast switchable hardware con-

texts make it cheaper to poll for information than to wait for an asynchronous signaling mechanism like an interrupt; (2) hardware contexts running on micro-engines are non-preemptible, thus the context must explicitly check for the presence of interrupt signal anyway; and (3) there exists no direct signaling path from host cores to micro-engines, so that such signals would have to be routed via the XScale core, resulting in prohibitive latency.

More specifically, every VIF is assigned two different bits in the host-side interrupt identifier register (one each for the send and receive directions). The bits are shared by multiple VIFs in case the total number of VIFs exceeds 8. Setting any bit in the identifier register causes a master PCI interrupt to be asserted on a host core. The interrupt service routine runs in the hypervisor context and using the association between bits and VIFs, can determine which VIF (or *potential set of VIFs* in case of sharing) generated the master interrupt, along with the reason, by reading the identifier register. Based on the reason (send/receive), an appropriate signal is sent to the guest domain associated with the VIF(s). This signal demultiplexing and forwarding (aka interrupt virtualization) is the sole involvement of the hypervisor in the I/O path.

3.2 Management responsibilites of the hypervisor

In this section, we discuss how the hypervisor assists the SV-NIC in providing isolation between VIFs. The SV-NIC isolates VIFs via spatial partitioning of resources, both on the device (such as SDRAM and micro-engine contexts) and on the host (the host memory). Memory resources belonging to a VIF are mapped into the address space of the guest domain owning that VIF by the hypervisor.

As discussed earlier, the send queue of a VIF exists in the NP's SDRAM, which is part of the host PCI address space. Access to it is available by default only to privileged domains, e.g., the controller domain. In order for a (non-privileged) guest domain to be able to access its VIF's send queue in this address space, the management driver uses Xen's *grant table mechanism* to authorize write access to the corresponding I/O memory region for the requesting guest domain. The guest domain can then request Xen to map this region into its page tables. Once the page table entries are installed, the guest domain can inject messages directly into the send queue.

In our current implementation, the host memory area that contains a VIF's receive queue is owned by the controller domain (dom0). The management driver grants access of the region belonging to a particular VIF to its corresponding guest domain. The guest domain then asks Xen to map this region into its page tables and can subsequently receive messages directly from the VIF's receive queue.

The above mappings are created once during VIF creation time and remain in effect for the life-time of the VIF (usually the life-time of its guest domain). All remaining logic to implement packet buffers inside the queues and the send/receive operations is implemented completely by the guest domain driver and on the NP micro-engines.

In summary, the hypervisor enforces isolation by restricting access to memory resources via page tables – a guest domain cannot access the memory space (neither upstream nor downstream) of VIFs other than its own.

3.3 SV-NIC vs. Driver-Domain approach to network virtualization

Current hypervisors use the 'driver-domain' approach to network virtualization. In this approach, henceforth termed HV-NIC, each guest domain is provided with one or more virtual 'front-end' network devices. Corresponding to each front-end device, a virtual 'back-end' network device is created in a privileged domain. This domain, terned 'driver-domain' can access the physical network device. The driver domain uses layer-2 software bridging to bridge virtual back-end devices and the physical network device.

Front-end and back-end network devices are connected via a point-to-point shared memory channel. In the *egress* path, a packet sent by the guest domain on front-end device requires the scheduling of driver-domain, where the packet is received by the back-end device driver and is forwarded to the software bridging code. Based on packet destination MAC address, bridging code may invoke the physical NIC's device driver to send the packet out. In the *ingress* path, a packet received by the physical NIC's device driver is forwarded to the bridging code, which may invoke a send on the back-end network device. This results in scheduling of destination guest domain to receive the packet on the front-end device. Xen uses a zero-copy approach, where a network packet is moved between guest domain and driver domain via *page flipping* [27]. As shown by Menon et al., the driver-domain approach requires substantial CPU cycles, negatively affects caches, and the virtualized network performance does not compare favorably against the network performance in native systems [25].

In our prototype setup, since the host does not have direct access to the gigabit network interface available on the ENP2611 board, the network processor on the board is used to tunnel network packets between the host and the interface. This provides to the host the illusion that the ENP2611 board is a gigabit ethernet interface. In fact, this tunnel interface is almost identical to a VIF. It contains two queues, a send-queue and a receive-queue, and it bears the ID of the physical ethernet interface. Its virtualization with the driver-domain approach discussed earlier comprises the HV-NIC implementation evaluated in this paper.

In comparison, the SV-NIC does not require the involvement of any domain other than the guest that is performing I/O. In addition, it removes most hypervisor interactions from the data fast path, by permitting each guest domain to directly interact with the virtualized device. This is done by exploiting device-level resources to map substantial virtualization functionality (i.e., the hypervisor functions and the device driver stack required to virtualize the device) to the processing resources located *close to* the physical network device.

We note that the total number of data copy and signaling operations performed by the host and the NP to access the physical device are the same for the SV-NIC accessed by a guest domain and for the tunnel interface accessed directly from the driver domain. Here, performance differences are due only to the effects from 'where' network virtualization code is run: inside a driver-domain or inside the device itself.

4. ENHANCING NETWORK I/O WITH HARDWARE I/O MMUS

Unlike the case of downstream access, where the host can

address any location in the NP's SDRAM, firmware restrictions in our prototype implementation platform limit the addressability of host memory by the NP to 64MB. In fact, even with firmware modifications, the hard limit is 2GB. Since the NP cannot access the complete host address space, all NP to host data transfers must target specific buffers in host memory, termed *bounce buffers*. This implies that the receive queue of the tunnel device or a VIF must consist of multiple bounce buffers.

In keeping with standard Unix implementations, the host-side driver copies the network packet from the bounce buffer in the receive queue to a socket buffer (*skb*) structure. An alternate approach avoiding this copy is to directly utilize receive queue memory for *skb*s. This can be achieved by either (1) implementing a specific skb structure and a new page allocator that uses the receive queue pages, or (2) instead of having a pre-defined receive queue, construct one that contains the addresses of allocated skbs. The latter effectively requires either that the NP is able to access the entire host memory (which is not possible due to the limitations discussed above) or that an I/O MMU is used for translating an I/O address accessible to the NP to the memory address of the allocated skb. For ease of implementation, we have not pursued (1) in our prototype, but it is an optimization we plan to consider in future work. Concerning (2), since our platform does not have a hardware I/O MMU, our implementation emulates this functionality by using bounce buffers plus message copying, essentially realizing a software I/O MMU. In future systems with hardware I/O MMU, this extra copy can be eliminated which will improve the performance of upstream NP accesses to host memory. Further, the hardware I/O MMU must be integrated with the self-virtualized device to enforce security isolation - a guest domain must be able to program I/O MMU with addresses for memory pages it owns and map them to I/O addresses for receive queue(s) for VIF(s) it owns.

5. EXPERIMENTAL EVALUATION

The experiments reported in this paper use two hosts, each with an attached ENP2611 board. The gigabit network ports of both boards are connected to a gigabit switch. Each host has an additional Broadcom gigabit ethernet card, which connects it to a separate subnet for developmental use.

Hosts are dual 2-way HT Pentium Xeon (a total of 4 logical processors) 2.80GHz servers, with 2GB RAM. The hypervisor used for system virtualization is Xen3.0-unstable [27]. Dom0 runs a paravirtualized Linux 2.6.16 kernel with a RedHat Enterprise Linux 4 distribution, while guest VMs run a paravirtualized Linux 2.6.16 kernel with a small ramdisk root filesystem based on the Busybox. The ENP2611 board runs a Linux 2.4.18 kernel with the MontaVista Preview Kit 3.0 distribution. Experiments are conducted with uniprocessor dom0 and guest VMs. Dom0 is configured with 512MB RAM, while each guest VM is configured with 32MB RAM. We use the default Xen CPU allocation policy, under which dom0 is assigned to the first hyperthread of the first CPU (logical CPU #0), and guest VMs are assigned one hyperthread from the second CPU (logical CPU #2 and #3). Logical CPU #1 is unused in our experiments. The Borrowed Virtual Time (bvt) scheduler with default arguments is the domain scheduling policy used for Xen.

Experiments are conducted to evaluate the costs and benefits of the SV-NIC in comparison with the driver-domain approach to network virtualization. Two sets of experiments are performed. The first set uses the HV-NIC, where the driver domain provides virtual interfaces to guest domains and uses the tunnel interface as the physical device. Our setup uses dom0 (i.e., the controller domain) as the driver domain. The second set of experiments evaluates the SV-NIC realization described in Section 3, which provides VIFs directly to guest domains without any driver domain involvement in the network I/O path.

The performance of the SV-NIC *vs.* the HV-NIC, i.e., of their virtual interfaces provided to guest domains, are evaluated with two metrics: latency and throughput. For latency, we wrote a simple libpcap [9] client server application, termed *psapp*, to measure the packet round trip times between two guest domains running on different hosts. The client sends 64-byte probe messages to the server using packet sockets and SOCK_RAW mode. These packets are directly handed to the device driver, without any Linux network layer processing. The server receives the packets directly from its device driver and immediately echoes them back to the client. The RTT serves as an indicator of the inherent latency of the network path.

For throughput, we use the iperf [7] benchmark application. The client and the server processes are run in guest domains on different hosts. The client sends data to the server over a TCP connection with buffer size set to 256KB (on guest VMs with 32MB RAM, linux allows only a maximum of 210KB), and the average throughput for the flow is recorded. The client run is repeated 20 times.

All experiments run on two hosts and use a 'n,n:1x1' access pattern, where 'n' is the number of guest domains on each host. Every guest domain houses one virtual network interface. On one machine, all guest domains on a machine run server processes, one instance per guest. On the second machine, all guest domains run client processes, one instance per guest. Each guest domain running a client application communicates to a distinct guest domain that runs a server application on the other host. Hence, there are a total n simultaneous flows in the system. In the experiments involving multiple flows, all clients are started simultaneously at a specific time in pre-spawned guest domains. We assume that the time in all guest domains is kept well-synchronized by the hypervisor.

5.1 Latency

The latency measured as the RTT by the psapp application includes both basic communication latency and the latency contributed by virtualization.

Another source of latency for the SV-NIC is the total number of signals sent per packet. On our platform, due to the limitations on interrupt identifier size, a single packet may require more than one guest domain to be signalled. Assuming these domains share the CPU, the overall latency, then, includes the time it takes to send a signal and *possibly*, the time spent on 'useless' work performed by a redundant domain. Whether or not such useless work occurs depends on the method for domain scheduling used on the shared CPU.

Using packet round trip time (RTT) as the measure of end-to-end latency, Figure 3 shows the RTT reported by *psapp* for the HV-NIC and the SV-NIC. On the x-axis is the total number of concurrent guest domains 'n' running on

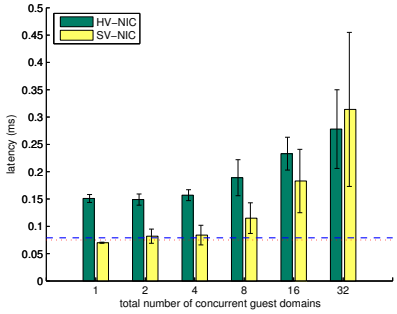

Figure 3: Latency of the HV-NIC and the SV-NIC. Dotted lines represent the latency for dom0 using the tunnel network interface in two cases: (1) No virtualization functionality (i.e., without software bridging), represented by fine dots, and (2) driver-domain functionality (i.e., with software bridging), represented by dash dots

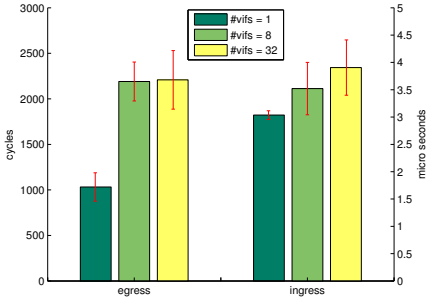

Figure 4: Latency microbenchmarks for the SV-NIC

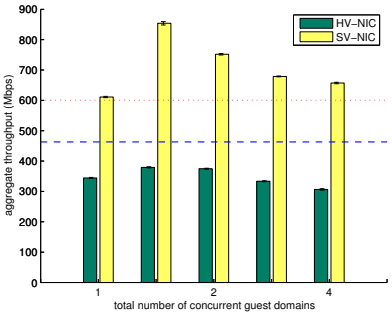

Figure 5: TCP througput for the HV-NIC and the SV-NIC. Dotted lines represent the throughput for dom0 using the tunnel network interface in two cases: (1) No virtualization functionality (i.e., without software bridging), represented by fine dots, and (2) driver-domain functionality (i.e., with software bridging), represented by dash dots

each host machine. On the y-axis is the *median* latency and inter-quartile range of the 'n' concurrent flows; each flow $i \in n$ connects $GuestDomain_i^{client}$ to $GuestDomain_i^{server}$. For each n, we combine N_i latency samples from flow $i, 1 \leq i \leq n$ as one large set containing $\sum_{i=1}^{n} N_i$ samples. The reason is that each flow measures *the same* random variable, which is end-to-end latency when n guest domains are running on both sides.

We use the median as a measure of central tendency since it is more robust to outliers (which occur sometimes due to unrelated system activity, especially under heavy load with many guest domains). Inter-quartile range provides an indication of the spread of values.

These results demonstrate that with the SV-NIC, it is possible to obtain close to a 50% latency reduction for virtual interfaces compared to Xen's driver-domain approach. This reduction results from the facts that dom0 is no longer involved in the network I/O path, and that hypervisor interactions are reduced. Also, with the SV-NIC, *the latency of using one of its VIFs in a guest VM is almost identical to using the HV-NIC's tunnel interface from the domain that has direct device access, dom0.* Our conclusion is that the basic cost of the self-virtualized device implementation is low.

The median latency value and inter-quartile range increase in all cases as the number of guest domains (and hence the number of simultaneous flows) increases. This is mostly because of increased CPU contention between guests and would not be present for many-core hosts, such as 32-way SMPs. In any case, the cost of virtualization at the SV-NIC remains low and shows good scalability, as shown by the microbenchmarks depicted in Figure 4. For brevity, these microbenchmarks are discussed in detail elsewhere [21].

Our prototype hardware's limitations in terms of interrupt identifier sizes leads to interrupt identifier sharing, which causes the latency of the SV-NIC to increase beyond that of the HV-NIC for 32 VIFs. This is because here, every identifier bit is shared among 4 VIFs, and hence, requires 1.5 redundant domain schedules on the average before a signal is received by the correct domain. On our system with only two CPUs available for guest VMs, these domain schedules also require context switching, which further increases latency. One component of this latency, the extra time taken by the hypervisor for interrupt virtualization, is demonstrated by the host-side microbenchmarks reported next. For $\#vifs = 1$, the host takes $\sim 1.99\mu s$ for interrupt virtualization. For $\#vifs = 32$, the average cost of interrupt virtualization increases to $\sim 11.57\mu s$. The cost of interrupt virtualization increases since for every interrupt, multiple domains need to be signaled, even redundantly in the case when $\#vifs > 8$. To address this issue, we are currently implementing a message based signaling mechanism, which will permit the SV-NIC to use a shared memory buffer to communicate to the hypervisor exactly which guest domain needs to be signaled, along with sending the master interrupt to the host. This will eliminate redundant signaling and thereby improve SV-NIC scalability.

5.2 Throughput

Figure 5 shows the mean and variance of the aggregate throughput of 'n' TCP flow(s) reported by iperf for the SV-NIC and the HV-NIC, where 'n' is the number of concurrent guest domains. The setup is similar to the latency experiment described above.

Based on these results, we make the following observations:

- *The performance of the SV-NIC is about 2X of that of*

host to NP	1088.1
NP to host	947.8

Table 1: Throughput of the PCI writes between the host and the NP (Mbps)

the HV-NIC, even for large numbers of guest domains. Several factors contribute to the performance drop for the HV-NIC, as discussed earlier.

- *The performance of using a single VIF in a guest VM with the SV-NIC is similar to using the HV-NIC (tunnel interface) in dom0 without any virtualization functionality.* This shows that the cost of I/O virtualization with the SV-NIC is low and that it purely resides in the device and the hypervisor.
- *The performance of the HV-NIC for any number of guests is always lower than with a single VIF in the SV-NIC.* The reason for this is the fact that the tunnel driver must enforce ordering over all packets and hence, it cannot take advantage of hardware parallelism effectively. In contrast, for the SV-NIC implementation, there is less contention for ordering as the number of VIFs increases. For example, on average 2 contexts will contend per VIF for ordering when #VIFs = 4, *vs.* 8 contexts, when #VIFs = 1 (or for the tunnel device in the HV-NIC case). Although a smaller number of contexts per VIF implies less throughput per VIF, the aggregate throughput for all the VIFs will be more when #VIFs > 1 *vs.* #VIFs = 1 (or for the tunnel device).

Further, the performance of the network I/O path for both SV-NIC and HV-NIC is limited by the upstream (NP to host) PCI performance. This is shown by the following microbenchmark. We measure the available throughput of the PCI path between the host and the NP for write (since PCI reads are avoided), by writing a large buffer across the PCI bus both from the host and from the NP. In order to model the behavior of packet processing in both SV-NIC and HV-NIC, the write is done 1500 bytes at a time. Also, for the upstream path, aggregate throughput is computed for 8 micro-engine contexts, where all of the contexts are copying data *without* any ordering requirement. This represents the best case performance available for network I/O from NP to host.

Results of this benchmark are presented in Table 1. They demonstrate that the bottleneck currently lies in the upstream path. This is further exacerbated by the ordering requirements and the need to use bounce buffers due to the limited addressability of host memory from the NP. Better upstream PCI performance can be achieved via the use of DMA engines available on the NP board. On the egress path, the host can provide data fast enough to the NP to sustain the gigabit link speed.

6. TOWARD SELF-VIRTUALIZED I/O IN FUTURE MANY-CORE SYSTEMS

A self-virtualized device, like the SV-NIC evaluated in this paper, exhibits high performance and scalability for I/O virtualization since it can exploit *multiple processing cores* situated *close* to the physical device and because these processing cores might be specialized for certain types of processing, e.g. micro-engines in a network processor. Further performance benefits can be obtained with better messaging fabrics connecting the self-virtualized device and guest domains executing on the host. Current architectural advances are addressing these issues:

- Large multi-core and many-core systems will provide the opportunity to utilize certain cores for virtualizing a particular peripheral. A highly optimized version of a driver-domain, one that implements just enough functionality and employs latency reducing techniques, such as polling rather than interrupts, can provide better performance and scalability than is possible in systems today.
- I/O devices will come 'closer' to compute cores, as evident from planned enhancements to system interconnects, such as Geneseo [2]. These will reduce the latencies of device accesses from cores, e.g. via cache-coherent data transfer between memory and I/O devices.
- Heterogeneous multi-core systems like IBM's Cell [10] processor or like general purpose hosts paired with network processors and other accelerators can provide specialized processing at lower costs than general purpose cores.
- Improved inter-core communication mechanisms, such as AMD's HyperTransport [3], will allow low latency signaling among cores. At the same time, shared caches and high-bandwidth memory interconnects will allow for faster data transfer among cores. Both of these properties will enable higher performance for the messaging fabric implemented via shared memory and inter-core signaling.

Based on these observations, we extend the self-virtualized device approach presented in this paper toward a hypervisor-level abstraction for high end, many-core systems, termed *self-virtualized I/O* (SV-IO). The SV-IO abstraction describes the resources used to implement device virtualization. Specifically, a device virtualization solution built with SV-IO is described to consist of (1) some number of processing components (cores), (2) a communication link connecting these cores to the physical device, a.k.a. the peripheral communication fabric, such as PCI Express enhanced with Geneseo, and (3) the physical device itself. By identifying these resources, *SV-IO can characterize, abstractly, the different implementation methods currently used to virtualize peripheral devices*, including those that fully exploit the multiple cores of modern computing platforms. Realizations of the SV-IO abstraction must efficiently map these components to available physical resources in order to obtain high performance and scalability.

Our ongoing and future work is using the SV-IO approach for several purposes. First, it is used to generalize the notion of SV-NIC presented in this paper to encompass all possible implementations of its functionality, on general purposes hosts in their driver domains and with alternative partitions of virtualization functionality across hosts and network processors. Second, it is used to describe virtualization for other devices, including high performance communication devices like SeaStar on Crays [13] or Infiniband [4], the latter already offering limited self-virtualization capabilities. Third, it extends the communication-centric treatment of self-virtualized devices pursued in this paper to one that addresses entirely different devices, including USB devices used for cameras [22], for example. Fourth, and most im-

portantly, with SV-IO, it becomes possible to flexibly and dynamically allocate platform resources to I/O virtualization tasks, so as to best meet application requirements in terms of acceptable overheads and desired performance.

7. RELATED WORK

Recent advancements in system virtualization techniques that offer high performance via para-virtualization have warranted re-evalutation of the applicability of virtualized platforms in the high performance computing domain [39]. Further optimizations in the management software stack, such as small, special purpose operating systems, are envisioned to provide further support to HPC on large many-core systems, both with and without virtualization [16, 17]. Similar to these developments, *self-virtualized devices* offload selected virtualization functionality from the hypervisor to the device itself, to enable high performance and scalable I/O virtualization. A concrete realization presented in the paper, the SV-NIC, provides direct, multiplexed, yet isolated, network access to guest domains via VIFs. Its philosophy is similar to fast messages [26] and U-Net [33], where network interfaces are provided to user space with OS-bypass. Since the VIF device driver for guest domains has the capability to map VIF resources to user space, the SV-NIC trivially incorporates these solutions. Similarly, new generation InfiniBand devices [23] offer functionality that is akin to this paper's ethernet-based SV-NIC, by providing virtual channels that can be directly used by guest domains. However, these virtual channels are less flexible than our SV-NIC in that no further processing can be performed on data at the device level.

In order to improve network performance for end user applications, multiple configurable and programmable network interfaces have been designed [37, 28]. These interfaces could also be used to implement a SV-NIC, as demonstrated by the CDNA prototype [36]. Another network device that implements this functionality for the zSeries virtualized environment is the OSA network interface [8], which uses general PowerPC cores for this purpose, in contrast to the NP used by our SV-NIC. We believe that using specialized network processing cores provides performance benefits for domain-specific processing, thereby allowing more efficient and scalable self-virtualized device implementations. Furthermore, these virtual interfaces can be efficiently enhanced to provide additional functionality, such as packet filtering and protocol offloading.

Although NPs have generally been used standalone for carrying out network packet processing in the fast path with minimal host involvement, previous work has also used them in a collaborative manner with hosts, to extend host capabilities, such as for fast packet filtering [14]. We use the NP in a similar fashion to implement the self-virtualized network interface. Application-specific code deployment on NPs and other specialized cores has been the subject of substantial prior work [30, 19, 38].

Modern computer systems perform a variety of tasks, not all of which are suitable for execution on general purpose processors. A platform with both general and specialized processing capabilities can provide the high performance required by specific applications. A prototype of such a platform is envisioned in [20] where a CPU and a NP are utilized in unison. More recently, multiple heterogeneous cores have been placed on the same chip [10]. Our work uses a similar heterogeneous platform, consisting of Xeon CPUs and an IXP2400 NP communicating via a PCI interconnect. This is in comparison to other solutions that use general purpose cores for network packet processing and other device-near tasks [29, 15].

The SV-IO abstraction bears resemblance to the virtual channel processor abstraction proposed in [24], although the intended use for virtual channel processors is to provide virtual devices for some system-level functionality, such as iSCSI. The purpose is to offload guests by not having them run their own iSCSI module which in turn communicates to the virtual network interface. VIFs provided by SV-IO can be similarly enhanced with added functionality. As a concrete example, we are currently constructing privacy enhanced virtual camera devices. Their *logical* VIFs provide an extension mechanism similar to *soft-devices* [34], the major difference being that we can flexibly choose where to extend the VIF, at the device level directly or at the host level. This flexibility is provided by utilizing both host- and device-resident processing components.

8. CONCLUSIONS AND FUTURE WORK

This paper advocates the self-virtualized device approach to virtualizing the 'smart' peripherals found in high end systems. It presents the design and implementation of a *self-virtualized network interface* (SV-NIC) using an IXP2400 network processor-based board. Performance of the virtual interfaces provided by the SV-NIC is analyzed and compared to the driver-domain approach (HV-NIC) used in commercial platforms and by the open-source Xen hypervisor. Experimental results demonstrate that the SV-NIC solution outperforms the HV-NIC approach. It also scales better with an increasing number of virtual interfaces used by an increasing number of guest domains.

The SV-NIC enables high performance in part because of its ability to reduce hypervisor involvement and eliminate driver-domain involvement in I/O path. In our solution, the hypervisor on the host is responsible for managing the virtual interfaces presented by the SV-NIC, but once a virtual interface has been configured, most actions necessary for network I/O are carried out without any hypervisor involvement.

Future work includes both improvements in the current SV-NIC implementation and enhancements to and generalizations of the SV-IO abstraction. In the short term, we will improve VIF performance in several ways:

- improve upstream VIF throughput by replacing micro-engine programmed I/O with DMA;
- improve TCP performance via TCP segment offload; and
- add support for large MTU sizes (jumbo frames).

In the longer term, we are investigating *logical* virtual devices built using SV-NICs and VIFs. A simple example is a VIF that provides additional services/programmability, such as packet filtering based on header information and application level filtering from message streams. Such logical virtual devices could also be enhanced with certain QoS attributes. An example QoS attribute pertinent to high performance systems is bounded guarantees on host computational resource utilization for I/O virtualization, which would provide a bound on the worst-case performance degradation a guest domain would experience. Based on this metric and the performance requirements for the guest, the sys-

tem can (1) provide performance guarantees for a domain in a virtualized environment, and (2) if performance requirements can no longer be met, use migration to move the guest to a platform where they can be met.

ACKNOWLEDGEMENTS

We are thankful to Jeffrey Daly at Intel and Kenneth McKenzie at Reservoir Labs for their persistent help with implementation issues regarding the 21555 bridge and the NP platform. The design of the SV-IO abstraction was motivated by research interactions with Jun Nakajima at Intel Corporation.

9. REFERENCES

[1] ENP-2611 Data Sheet. http://www.radisys.com/files/ENP-2611_07-1236-05_0504_datasheet.pdf.
[2] The geneseo proposal to enhance pci-express. http://www.intel.com/pressroom/archive/releases/20060927comp_a.htm.
[3] Hypertransport interconnect. http://www.hypertransport.org/.
[4] Infiniband interface. http://www.infinibandta.org/home.
[5] Intel 21555 Non-transparent PCI-to-PCI Bridge. http://www.intel.com/design/bridge/21555.htm.
[6] Intel Virtualization Technology Specification for the IA-32 Intel Architecture. ftp://download.intel.com/technology/computing/vptech/C97063-002.pdf.
[7] Iperf. http://dast.nlanr.net/projects/Iperf.
[8] OSA-Express for IBM eserver zSeries and S/390. www.ibm.com/servers/eserver/zseries/library/specsheets/pdf/g2219110.pdf.
[9] Tcpdump/libpcap. http://www.tcpdump.org/.
[10] The Cell Architecture. http://www.research.ibm.com/cell/.
[11] The VMWare ESX Server. http://www.vmware.com/products/esx/.
[12] Intel IXP2400 Network Processor: Hardware Reference Manual, October 2003.
[13] R. Alverson. Red storm: A 10,000 node system with reliable, high bandwidth, low latency interconnect. In *Proc. of Hot Chips 13*, 2003.
[14] H. Bos, W. de Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis. Ffpf: Fairly fast packet filters. In *Proc. of OSDI*, 2004.
[15] T. Brecht et al. Evaluating Network Processing Efficiency with Processor Partitioning and Asynchronous I/O. In *Proc. of EuroSys*, 2006.
[16] P. G. Bridges, A. B. MacCabe, and O. Krieger. System Software for High End Computing. *SIGOPS Oper. Syst. Rev.*, 40(2), 2006.
[17] S. Chakravorty, C. L. Mendes, L. V. Kale, T. Jones, A. Tauferner, T. Inglett, and J. Moreira. HPC-Colony: services and interfaces for very large systems. *SIGOPS Oper. Syst. Rev.*, 40(2), 2006.
[18] B. Dragovic et al. Xen and the Art of Virtualization. In *Proc. of SOSP*, 2003.
[19] A. Gavrilovska et al. Stream Handlers: Application-specific Message Services on Attached Network Processors. In *Proc. of HOT-I*, 2002.
[20] F. T. Hady et al. Platform Level Support for High Throughput Edge Applications: The Twin Cities Prototype. *IEEE Network*, July/August 2003.
[21] Himanshu Raj and Ivan Ganev and Karsten Schwan and Jimi Xenidis. Self-Virtualized I/O: High Performance, Scalable I/O Virtualization in Multi-core Systems. Technical Report GIT-CERCS-06-02, CERCS, Georgia Tech, 2006.
[22] J. Kong, I. Ganev, K. Schwan, and P. Widener. Cameracast: Flexible access to remote video sensors. In *Multimedia Computing and Networking (MMCN'07)*, San Jose, CA, USA, Jan. 2007.
[23] J. Liu, W. Huang, B. Abali, and D. K. Panda. High Performance VMM-Bypass I/O in Virtual Machines. In *Proc. of USENIX ATC*, 2006.
[24] D. McAuley and R. Neugebauer. A case for Virtual Channel Processors. In *Proc. of the ACM SIGCOMM 2003 Workshops*, 2003.
[25] A. Menon et al. Diagnosing performance overheads in the xen virtual machine environment. In *Proc. of VEE*, 2005.
[26] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. In *Proc. of Supercomputing*, 1995.
[27] I. Pratt et al. Xen 3.0 and the Art of Virtualization. In *Proc. of the Ottawa Linux Symposium*, 2005.
[28] I. Pratt and K. Fraser. Arsenic: A User Accessible Gigabit Network Interface. In *Proc. of INFOCOM*, 2001.
[29] G. Regnier et al. ETA: Experience with an Intel Xeon Processor as a Packet Processing Engine. *IEEE Micro*, 24(1):24–31, 2004.
[30] M. Rosu, K. Schwan, and R. Fujimoto. Supporting Parallel Applications on Clusters of Workstations: The Virtual Communication Machine-based Architecture. In *Proc. of Cluster Computing*, 1998.
[31] J. H. Salim, R. Olsson, and A. Kuznetsov. Beyond softnet. In *Proc. of 5th USENIX Annual Linux Showcase and Conference*, 2001.
[32] V. Uhlig et al. Towards Scalable Multiprocessor Virtual Machines. In *Proc. of the Virtual Machine Research and Technology Symposium*, 2004.
[33] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: a user-level network interface for parallel and distributed computing. In *Proc. of SOSP*, 1995.
[34] A. Warfield, S. Hand, K. Fraser, and T. Deegan. Facilitating the development of soft devices. In *Proc. of USENIX ATC*, 2005.
[35] R. West et al. Dynamic window-constrained scheduling of real-time streams in media servers. *IEEE Transactions on Computers*, 2004.
[36] P. Willmann et al. Concurrent Direct Network Access for Virtual Machine Monitors. In *Proc. of HPCA*, 2007.
[37] P. Willmann, H. Kim, S. Rixner, and V. Pai. An Efficient Programmable 10 Gigabit Ethernet Network Interface Card. In *Proc. of HPCA*, 2005.
[38] H. youb Kim and S. Rixner. Tcp offload through connection handoff. In *Proc. of EuroSys*, 2006.
[39] L. Youseff, R. Wolski, B. Gorda, and C. Krintz. Evaluating the Performance Impact of Xen on MPI and Process Execution For HPC Systems. In *Proc. of International Workshop on Virtualization Technologies in Distributed Computing (VTDC)*, 2006.