Self-Virtualized I/O: High Performance, Scalable I/O Virtualization in Multi-core Systems

Himanshu Raj
Ivan Ganev
Karsten Schwan
CERCS, College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{rhim,ganev,schwan}@cc.gatech.edu
Jimi Xenidis
IBM T.J. Watson Research Lab
Yorktown, NY
jimix@watson.ibm.com

Abstract

Virtualizing I/O subsystems and peripheral devices is an integral part of system virtualization. This paper advocates the notion of *self-virtualized I/O* (SV-IO). Specifically, it proposes a hypervisor-level abstraction that permits guest virtual machines to efficiently exploit the multi-core nature of future machines when interacting with virtualized I/O. The concrete instance of SV-IO developed and evaluated herein (1) provides virtual interfaces to an underlying physical device, the network interface, and (2) manages the way in which the device's physical resources are used by guest operating systems. The performance of this instance differs markedly depending on design choices that include (a) how the SV-IO abstraction is mapped to the underlying host- *vs.* device-resident resources, (b) the manner and extent to which it interacts with the HV, and (c) its ability to flexibly leverage the multi-core nature of modern computing platforms. A *device-centric* SV-IO realization yields a *self-virtualized network device* (SV-NIC) that provides high performance network access to guest virtual machines. Specific performance results show that for high-end network hardware using an IXP2400-based board, a virtual network interface (VIF) from the device-centric SV-IO realization provides ∼77% more throughput and ∼53% less latency compared to the VIF from a *host-centric* SV-IO realization. For 8 VIFs, the aggregate throughput (latency) for device-centric version is 103% more (39% less) compared to the host-centric version. The aggregate throughput and latency of the VIFs scales with guest VMs, ultimately limited by the amount of physical computing resources available on the host platform and device, such as number of cores. The paper also discusses architectural considerations for implementing self-virtualized devices in future multi-core systems.

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 for x86-based machines [7]. In all such cases, the basis for machine virtualization are the hypervisors (HVs) 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 [17] and VMWare ESX server [12].The virtualization methods

used include resource partitioning, time sharing, or a combination thereof. These methods are used to: create virtual instances of all physical resources, including the peripheral devices attached to the host, and dynamically manage virtualized components among the multiple guest domains in a manner that enforces both physical and performance isolation between these domains.

This paper focuses on I/O device virtualization, by presenting the abstract notion of *Self-Virtualized I/O* (SV-IO). SV-IO captures all of the functionality involved in virtualizing an arbitrary peripheral device. It offers virtual interfaces (VIFs) and an API with which guest domains can access these interfaces. It specifies that the actual physical device must be multiplexed and demultiplexed among multiple virtual interfaces. It states that such multiplexing must ensure performance isolation across the multiple domains that use the physical device, and/or meet QoS requirements from guest domains stated as fair share or other metrics of guaranteed performance.

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, and (3) the physical device itself. By identifying these resources, *SV-IO can characterize, abstractly, the diverse implementation methods currently used to virtualize peripheral devices*, including those that fully exploit the multiple cores of modern computing platforms. These methods, characterized as *host-centric* methods since all virtualization functionality executes on host processing cores, include using a driver domain per device [34], using a driver domain per one set of devices [28], or running driver code as part of the HV itself [17]. The latter approach has been dismissed in order to avoid HV complexity and increased probability of HV failures caused by potentially faulty device drivers. Therefore, current systems favor the former approaches, but performance suffers from the fact that each physical device access requires the scheduling and execution of multiple domains. The SV-IO abstraction facilitates alternative, *device-centric*, realizations that address this issue, using metrics that include both the scalability of virtualization and the raw performance of individual virtualized devices. *A device-centric SV-IO realization implements selected virtualization functionality on the device itself*, resulting in less host involvement and potential performance benefits.

To demonstrate the utility of the device-centric SV-IO realization, we have created a self-virtualized network device (SV-NIC) on an implementation platform comprised of an IA-based host and an IXP2400 network processor-based gigabit ethernet board, using the Xen HV [17]. Since the IXP2400 network processor contains multiple processing elements situated *close* to the physical I/O device, this device-centric SV-IO realization efficiently exploits these resources to offer levels of performance exceeding that of the host-centric realizations used in existing systems. In particular, the self-virtualized network device (SV-NIC):

1. exploits IXP-level resources by mapping substantial virtualization functionality (*i.e.*, the HV functions and the device stack required to virtualize the device) to multi-core resources located *near* the physical network device,
2. removes most HV interactions from the data fast path, by permitting each guest domain to directly interact with the virtualized device, and
3. avoids needless transitions across the (relatively slow) PCI-based communication link between host and device.

The implementation also frees host computational resources from simple communication tasks, thereby permitting them to be utilized by guest VMs and improving platform scalability to permit a larger number of virtual machines.

The scalability of device virtualization solutions constructed with the SV-IO abstraction depends 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 at the discretion of SV-IO, such as the physical communication bandwidth and the maximum number of processing cores that can be deployed by the SV-IO. Results also show how certain design choices made in the SV-NIC implementation of SV-IO are affected by, and/or suggest the utility of, specific architectural features of modern computing platforms. One example is the necessity of integrating I/O MMU support with SV-IO.

In summary, this paper makes the following technical contributions:

1. It presents the SV-IO abstraction for I/O virtualization and outlines multiple design choices for realizing the abstraction on current platforms.
2. Among the design choices evaluated are a *host-centric* and a *device-centric* realization of SV-IO for a high end network device. By using this device's internal multi-core resources, concurrency in the device-centric implementation of SV-IO results in scalability for virtual interfaces (VIFs), along with high bandwidth and low end-to-end latency.
3. It explores some of the architectural implications that affect the performance of SV-IO realizations for future multi-core platforms.

The purpose of SV-IO support for both device- and host-centric implementations of device virtualization is to give system developers the flexibility to make choices suitable for specific target platforms. Factors to be considered in such choices include actual host *vs.* device hardware, host- *vs.* device-level resources, the communication link between them, and system and application requirements. In fact, evidence exists for both host- and device-centric solutions. The former represents a current industry trend that aims to exploit general multi-core resources. The latter is bolstered by substantial prior research, with examples including intelligent network devices [32, 19, 40], disk subsystems [31, 4], and even network routers [38], with recent work focusing on network virtualization [3].

Performance results demonstrate that the SV-IO abstraction meets its joint goals of high performance and flexibility in implementation. For a platform with a high end network device, for example, we show that the device-centric realization of SV-IO results in an SV-NIC that permits virtual devices to operate at full link speeds for 100 Mbps ethernet links. At gigabit link speeds, PCI performance dominates the overall performance of virtual devices. A VIF from this SV-NIC provides TCP throughput and latency of $\sim$ 620Mbps and $\sim$ .076ms, respectively, which is $\sim$77% more throughput and $\sim$53% less latency when compared to a VIF from a *host-centric* SV-IO realization. Finally, performance scales well for both host- and device-centric SV-IO realizations with an increasing number of virtual devices, one device per guest domain, although the device-centric realization performs better. For example, for 8 VIFs, the aggregate throughput (latency) for the device-centric version is 103% more (39% less) compared to the host-centric version.

In the remainder of this paper, Section 2 introduces the SV-IO abstraction and its components. Section 3 describes various design choices available in modern computing platforms to realize the SV-IO abstraction. Section 4 describes the design and functionality of a concrete SV-IO realization for a high-end network device, including both host- and device-centric versions, followed by platform specific implementation details of our prototype in Section 5. Experimental evaluations in Section 6 present comparative performance results for host- and device-centric SV-IO realizations. Section 7 presents some architectural considerations for SV-IO realizations in future multi-core systems, followed by related work. Section 9 concludes the paper and outlines future research.

2 The SV-IO Abstraction

Self-virtualized I/O (SV-IO) is a *hypervisor-level abstraction designed to encapsulate the virtualization of I/O devices*. Its goals are:

- scalable multiplexing/demultiplexing of a large number of *virtual devices* mapped to a single physical device,
- providing a lightweight API to the HV 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 future hardware platforms.

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

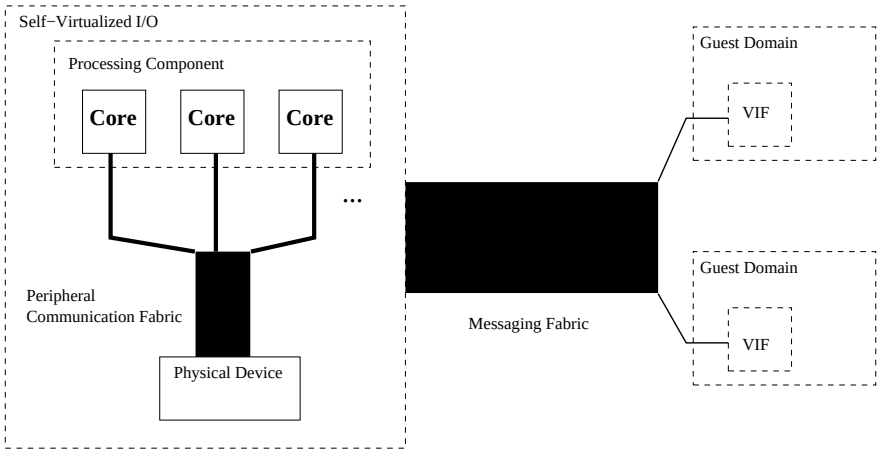


Figure 1: SV-IO Abstraction

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 SV-IO that the guest has enqueued a message in the send queue. The other signal is used by the SV-IO 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 SV-IO are interchanged. A particular implementation of SV-IO need not use all of these defined signals. For example, if the SV-IO polls the send queue to check the availability of message from the guest domain, it is not required to send the signal from guest domain to the SV-IO. Furthermore, queue signals are configurable at runtime, so that they are only sent when expected/desired from the other end. For example, a network driver using NAPI [33] does not expect to receive any interrupts when it processes the receive for a bunch of incoming network packets.

2.2 SV-IO Design

The SV-IO abstraction has four logical components, as depicted in Figure 1. The *processing component* consists of one or more *cores*. This component is connected to the *physical I/O device* via the *peripheral communication fabric*. Guest domains communicate with the SV-IO using VIFs and via the *messaging fabric*.

The two main functions of SV-IO 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 SV-IO's processing component. The processing component then performs all required processing on 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 SV-IO 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.* DWCS [37], 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 Realizing SV-IO: Design Choices

Modern computing platforms' rich architectural resources provide many design choices when realizing components of the SV-IO abstraction:

- The peripheral communication fabric connecting the processing component to the physical device could be a dedicated/specialized interconnect, such as the media and switch fabric (MSF) that is used to connect the IXP2400's network processor cores to the physical network port [13], or it could be a shared interconnect like PCI or HyperTransport.
- The messaging fabric connecting the processing component to guest domains could be realized using shared memory (with/without coherence), an interconnect like PCI, or a combination thereof, depending on the locations of *cores* in the SV-IO's processing component.
- Cores in the processing component could be heterogeneous or homogeneous, in contrast to the homogeneous cores used by guest domains. For example, rather than using all IA32-based cores as done by the guest domains, SV-IO could be mapped to specialized processor cores designed for network I/O processing. An advantage of specialized cores is that they need not offer the multitude of resources and features present in general cores, thereby saving chip real-estate while still providing comparable or even improved performance [22]. Another advantage is improved platform power efficiency. Potential disadvantages of heterogeneous cores are well known. One is the cost of implementing I/O subsystem software on platforms with different instruction sets and requiring different programming methods. Another is their comparative inflexibility compared to general cores, making it difficult to implement more complex functionality [40] there.
- Another choice for the processing component is to use dedicated cores to realize SV-IO, or to multiplex such I/O functions on cores shared with other processing activities. For high performance systems, there is evidence of performance advantages, due to reduced OS 'noise', derived from at least temporally dedicating certain cores to carry out I/O *vs.* computational tasks [27].

SV-IO Implementations

Our SV-IO implementations utilize the Xen hypervisor. In Xen, the standard implementation of device I/O uses *driver domains*, which are special guest domains that are given direct access to physical devices via some physical interconnect (*e.g.*, PCI). The driver domain provides the virtual interfaces, *e.g.*, a virtual block device or a virtual network interface, to other guest domains. The driver domain also implements the multiplex/demultiplex logic for sharing the physical device among virtual interfaces, the logic of which depends on the properties of each physical device. For instance, time sharing is used for the network interface, while space partitioning is used for storage. The hypervisor schedules the driver domains to run on general purpose host cores.

While Xen is not currently structured using SV-IO, the functions it runs in the driver domain for each physical device being virtualized are equivalent to those of an SV-IO-based device. Host cores belonging to the driver domain are the SV-IO's processing components, and they are architecturally homogeneous to the cores running guest domains. Host cores also run the SV-IO components that provide its management and

I/O functionality. The peripheral communication fabric is implemented via the peripheral interconnect, *e.g.*, PCI. The messaging fabric to communicate between cores running the driver domain and guest domains is implemented via shared memory. The sharing of cores used by the processing component is dependent on the hypervisor's scheduling policy. We refer to this approach as a *host-centric* realization of SV-IO, since all virtualization logic executes on host cores.

A *device-centric* realization of SV-IO exploits the processing elements 'close to' physical devices, such as processing elements in network processor-based platforms [2] and in SCSI adapters [4]. In a device-centric realization, the interconnect between the physical I/O device and on-device processing elements form the peripheral communication fabric, *e.g.* MSF, while the interconnect between the host system and the high end I/O device forms the messaging fabric, *e.g.*, PCI. Performance and/or scalability for the SV-IO device are improved when it is possible to better exploit the device's processing resources, to improve device behavior due to 'fabric near' control actions [32], or to shorten the path from device to guest domain. A specific example of a device-centric SV-IO realization is presented in the next section.

We choose the terms *device- or host-centric to refer to the location(s) of the majority rather than the entirety of SV-IO processing functionality.* Our SV-NIC implementation, for instance, requires host assistance for certain control plane device/guest interactions. Similarly, host-centric SV-IO will require some degree of device-level support, *e.g.*, the capability to perform I/O.

4 SV-IO Realizations for High-End Network Interface Virtualization

4.1 Hardware Platform and Basic Concepts

The communication device used is an IXP2400 network processor(NP)-based RadiSys ENP2611 board [2]. This resource-rich network processor features a XScale processing core and 8 RISC-based specialized communication cores, termed *micro-engines*. Each micro-engine supports 8 hardware contexts with minimal context switching overhead. The physical network device on the board is a PM3386 gigabit ethernet MAC connected to the network processor via the *media and switch fabric* (MSF) [13]. The board also contains substantial memory, including 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 combined host-NP platform represents one point in the design space of future multi-core systems, offering heterogeneous cores for running applications, guest OSs, and I/O functionality. As will be shown later, the platform is suitable for evaluating and experimenting with the scalability and with certain performance characteristics of the SV-IO abstraction, but it lacks the close coupling between host and NP resources likely to be found in future integrated multi-core systems. Specifically, in our case, the NP resides in the host system as a PCI add-on device, and it is connected to the host PCI bus via the Intel 21555 non-transparent PCI bridge [5]. This bridge allows the NP to only access a portion of host RAM resources via a 64MB PCI address window. In contrast, in the current configuration, host cores can access all of the NP's 256MB of DRAM.

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.

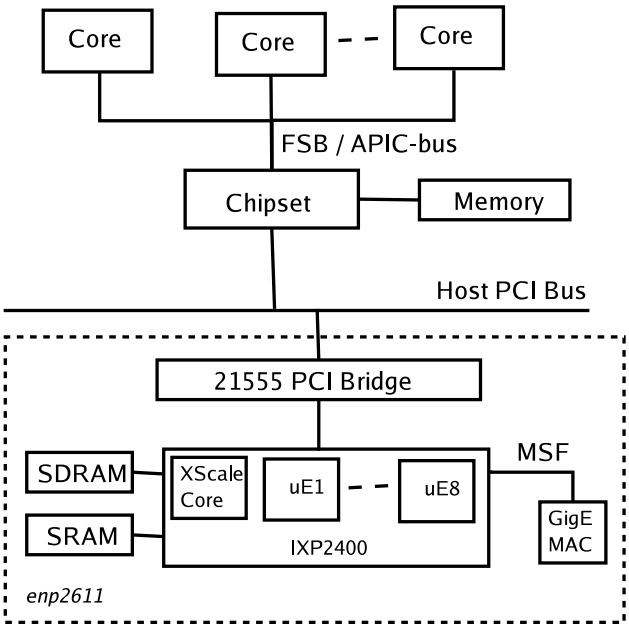


Figure 2: Host-NP Platform

4.2 Host-Centric Implementation of SV-IO

As explained in the previous section, in a host-centric realization of SV-IO, the network interface's virtualization logic runs in the driver domain (or controller domain) on host cores. The processing power available on the NP is used to tunnel network packets between the host and the gigabit ethernet interface residing on the board. 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. These queues contain a ring structure for queue maintenance and the actual packet buffers.

The NP's XScale core is not involved in the data fast path. Its role is to carry out control actions, such as starting and stopping the NP's micro-engines. The data fast path, *i.e.*, performing network I/O, is solely executed by micro-engines. In particular, a single micro-engine thread polls the send-queue and sends out packets queued by the device driver running in the driver domain onto the physical port. In case the driver domain fills up the entire queue before the micro-engine thread services it, the driver domain requests a signal to be sent when further space in the send-queue is available. The micro-engine thread sends this signal after it has processed some packets from the send-queue. A second micro-engine's execution contexts are used for receive-side processing – they select the packets from the physical interface and enqueue them on the receive-queue, in order. For each packet enqueued, a signal is sent to the driver domain, if required. The host side driver for the tunnel interface uses NAPI, which may disable this signal to reduce the signal processing load on the host in case the packet arrival rate is high. Thus, *the signals are only sent by the NP to driver domain.* Both signals are implemented as different identifier bits of the host-side doorbell register; the IRQ handler running in the driver domain determines the type of signal based on the identifier bit.

Software ethernet bridging, virtual network interfaces, front-end device drivers in guest domains, and back-end device drivers in the driver domain are used to virtualize this tunnel device. Xen's network interface virtualization is described in detail in [28].

4.3 SV-NIC: Device-Centric Implementation

In our NP-based, device-centric implementation, termed SV-NIC, most of the processing component, the peripheral communication fabric, and the physical I/O device components of the SV-IO abstraction are

situated on the ENP2611 board itself. The board's programmability and substantial processing resources make it easy to experiment with alternative SV-NIC implementation methods, as will become evident in later sections.

The SV-NIC implementation uses the following mapping for SV-IO components. The processing component is mapped to the XScale core and the micro-engines available on the board, along with one or more host processing cores. The peripheral communication fabric consists of the *Media and Switch Fabric* (MSF) [13]. The physical I/O device, the PM3386 gigabit ethernet controller, connects to the network processor via MSF. The processing component uses PCI as the messaging fabric to communicate with the guest domains via the virtual interface (VIF) abstraction.

The SV-NIC directly exports its VIF abstraction to guest domains as virtual network device. The send queue of each VIF is used for outgoing packets from guest domains and the receive queue is used for incoming packets to guest domains. As explained in more detail in the next subsection, 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 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 SV-IO is slower than the host processor). Similarly, the receive code of the guest domain driver uses the NAPI 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 SV-NIC and guest domain.

4.3.1 Functionality breakdown of processing components for SV-NIC

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 3 depicts various *management interactions* between the SV-NIC's processing components and the guest domain to *create a VIF*. 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). Setup and usage of these paths is deferred to Section 4.3.2, since it is dependent on various techniques employed by the Xen hypervisor.

Other management functionality includes the *destruction of VIF* and *changing attributes* of a VIF or of 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 NICs 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 that execute on different processing components of the SV-NIC. The host-side driver is part of the OS running in the controller domain (dom0). It runs on the host core(s). 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.

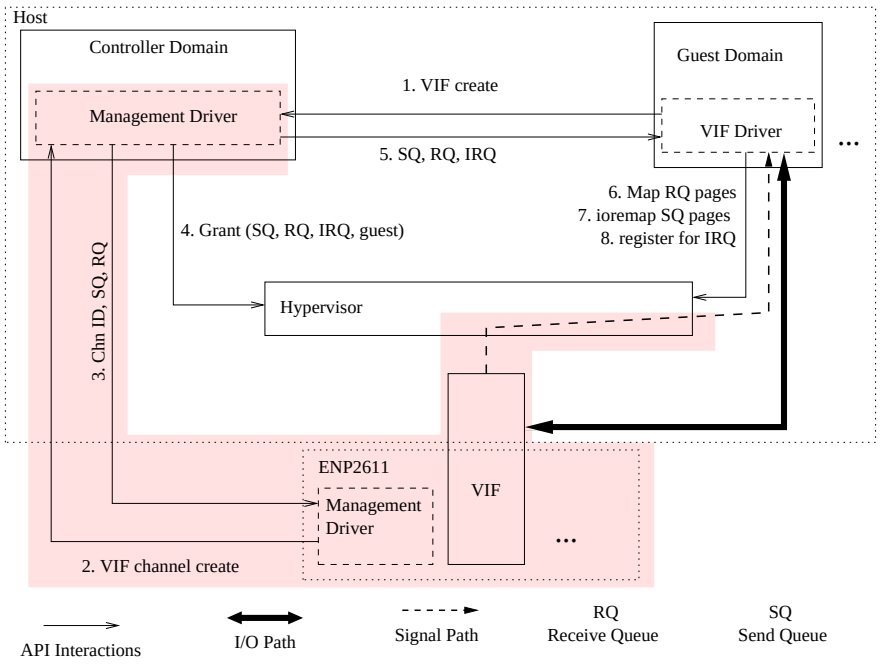


Figure 3: Management Interactions between SV-NIC, hypervisor and the guest domain to create a VIF. Shaded region depicts the boundary of SV-IO abstraction.

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 physical device; and
- *ingress:* demultiplex the packets received from the physical 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.

In our current implementation, *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 contexts 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 for a signal from the *previous* context in order 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.

Our 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 contexts 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 the host core(s) of SV-IO's processing component. Using the association between bits and VIFs, the SV-NIC 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 functionality of SV-IO is implemented as part of the Xen hypervisor itself.

In our current implementation, the master PCI interrupt generated by SV-NIC is sent to a specific host core. This core runs the signal demultiplexing and forwarding (aka interrupt virtualization) in hypervisor context. Thus, the set of host cores, which is a part of SV-IO's processing component, includes the cores assigned for the controller domain and the core performing interrupt virtualization.

4.3.2 Management Role of the Xen HV

Our device-centric realization of SV-IO, the SV-NIC, provides VIFs directly to guest domains. There is *minimal* involvement of the HV, and little additional host-side processing is required for I/O virtualization. The previous section described the interrupt virtualization role played by the HV for the SV-NIC. This section describes the management role of the HV in the setup phase of a VIF.

In order for a guest domain to utilize the VIF provided by the SV-NIC, it must be able to:

- *write messages* in the NP SDRAM corresponding to the *VIF send queue*; and
- *read messages* from the host RAM corresponding to the *VIF receive queue*.

The NP's SDRAM 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. For security reasons, the ring structure part of this region is read-only mapped for the guest, while the other part containing the packet buffers is mapped read-write. This is necessary because if the ring structure was writable, a malicious guest could influence the NP to read from arbitrary locations and inject bogus packets on the network.

In our current implementation, the host memory area accessible to the NP is owned by the controller domain. 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 part of this region containing the ring structure is mapped read-only, while the part containing actual packet buffers is mapped read-write. 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.

The ring structure is mapped read-only so that a malicious guest cannot influence the NP to perform writes to memory areas it does not own, thereby corrupting other domain's state. This is similar to the issue of DMA security isolation: if a guest domain is allowed to program DMA addresses in a device, then it can program it to an area of memory that it may not own, thereby corrupting the hypervisor's or other domain's memory. In Section 5.2, we describe how this issue can be addressed with future hardware I/O MMUs.

In summary, the grant table mechanism described above enforces security isolation – a guest domain cannot access the memory space (neither upstream nor downstream) of VIFs other than its own. Also, since a guest domain cannot perform any management related functionality, it cannot influence the NP to perform any illegal I/O to a VIF that it does not own.

4.3.3 Alternate SV-NIC Implementation

Currently, the host core contributing to the SV-NIC's processing, *i.e.*, performing interrupt virtualization, is also used for other host activities, as determined by the hypervisor's scheduling decisions. In future *many core* systems, an alternate design choice would be to dedicate a host core to this task. The final version of this paper will present comparative performance results including this alternate SV-NIC implementation.

Beyond its ability to exploit the resources of future many-core machines, we have multiple reasons for pursuing this alternate SV-NIC realization. First, such an implementation would improve performance isolation for the SV-NIC. One example is a signal sent by the micro-engines for a VIF whose corresponding guest domain is not currently scheduled to run on that host core. In our current implementation, such a signal unnecessarily preempts the currently running guest domain to the 'Xen context' for interrupt servicing. In contrast, our alternate implementation will use the host core exclusively for interrupt virtualization and hence, would not need to interrupt any guest domain. Further, it may be possible to abandon signaling via PCI interrupts altogether, since the dedicated host core can poll for signals as messages from the NP. This would reduce the latency of the signaling path.

There are also negative tradeoffs associated with the proposed alternate SV-NIC. One tradeoff is that while attaining latency improvements, a non-interrupt-based approach will cause wasted cycles and energy due to CPU spinning. This tradeoff is discussed in Section 7.2 in more detail. Another negative element of the approach is that all signals must always be forwarded by the SV-IO host core to the core running the guest domain as an inter-processor interrupt (IPI). That is, in this case, it is not possible to opportunistically make an upcall from the SV-IO host core to the guest domain in case the intended guest domain is currently scheduled to run on the same host core.

5 Platform-Specific Implementation Detail and Insights

In this section, we discuss some platform-specific limitations and how our current SV-IO realizations deal with them, along with insights on improvements possible with certain enhancements.

5.1 PCI Performance Limitations

Key requirements for the virtualized network device are (1) low communication overhead and (2) high performance in terms of bandwidth and latency. To attain both, our VIF and tunnel device implementations must deal with certain limitations of our chosen host/NP implementation platform. One challenge is to deal with the platform's limitations on PCI read bandwidth, as shown by a microbenchmark in Section 6.3.2. Toward this end, the current implementation avoids PCI reads whenever possible, by placing the send message queue into the NP's SDRAM (the *downstream communication space*), while the receive message queue is implemented in host memory (the *upstream communication space*). As a result, both on egress and ingress paths, PCI reads are avoided by guest domains and by SV-IO's processing components since relevant information is available in *local* memory.

5.2 Need for I/O MMU

Unlike the case of downstream access, where the host can address any location in the NP's SDRAM, current firmware restrictions limit the addressability of host memory by the NP to 64MB. 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*. The receive queue of the tunnel device or a VIF consists of multiple bounce buffers. For ease of implementation, all bounce buffers are currently allocated contiguously in host memory, but this is not necessary with this hardware.

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 *bus 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 a bus 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 summary, *the construction of the network I/O path would be facilitated by efficient upstream translated NP accesses to host memory.*

A related issue for the SV-NIC is to provide performance and security isolation among multiple VIFs. Our implementation attains performance isolation on the NP itself by spatially partitioning memory resources and time-sharing the NP's micro-engine hardware contexts. Some aspect of security isolation is provided by the host hypervisor, as discussed in the previous section. We next discuss the role of I/O MMUs in the context of security isolation.

For security isolation in the upstream network I/O path, we rely on the fact that the ring structure in the receive queue of a VIF is immutable to the guest. This requirement can be relieved by having the NP perform run-time checks to ensure that the bus address provided by the guest on the receive queue ring refers to a memory address that indeed belongs to the guest domain. These runtime checks can remove the need for bounce buffers in case the device can access all host memory, which may be facilitated by a hardware I/O MMU. Performing these checks is straightforward when the hypervisor statically partitions the memory among the guest domains, since it reduces to a simple range check. However, a domain can have access to certain memory pages that are "granted" to it by other domains, thereby implying that a runtime check must also search the grant table of the guest domain owning the VIF. Another approach would be for the hypervisor to provide a *map*, or bit vector of all of the memory pages currently owned by a guest domain. Based on this map, either the self-virtualized device or the hardware I/O MMU can decide whether an upstream I/O transaction takes place, depending on whether the target bus address is owned by the guest domain. Recent I/O MMUs available with hardware-assisted virtualization technology, such as AMD's Pacifica [1] or Intel's VT-D [14], support similar functionality, albeit for providing exclusive access of a single I/O device to a guest domain. These I/O MMUs must be enhanced to include multiple virtual devices per physical device in order to be useful with self-virtualized devices.

6 Performance Evaluation

6.1 Experiment Basis and Description

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 [28]. Dom0 runs a paravirtualized Linux 2.6.16 kernel with a RedHat Enterprise Linux 4 distribution, while domUs run a paravirtualized Linux 2.6.16 kernel with a small ramdisk root filesystem based on the Busybox distribution. 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 domUs. Dom0 is configured with 512MB RAM, while each domU 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 domUs 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 host- *vs.* device-centric SV-IO realizations, for virtualized hosts. For the sake of brevity, we nickname these realizations HV-NIC and SV-NIC, respectively. Two sets of experiments are performed. The first set uses HV-NIC, where the driver domain provides virtual interfaces to guest domains. Our setup uses dom0 (*i.e.*, the controller domain) as the driver

domain. Using the host-centric approach as the base case, the second set of experiments evaluates the SV-NIC realization described in Section 4.3, which provides VIFs directly to guest domains without any driver domain involvement in the network I/O path.

The performance of SV-NIC *vs.* HV-NIC realizations, *i.e.*, of their virtual interfaces provided to guest domains, are evaluated with two metrics: latency and throughput. For latency, a simple libpcap [10] client server application, termed *psapp*, is used 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 client sends a probe packet to the server and waits indefinitely for the reply. After receiving the reply, it waits for a random amount of time, between 0 and 100ms, before sending the next probe. The RTT serves as an indicator of the inherent latency of the network path.

For throughput, we use the iperf [8] benchmark application. The client and the server processes are run in guest VMs on different hosts. The client sends data to the server over a TCP connection with buffer size set to 256KB (on domUs 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 VIF. 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 (with resolution at 'second' granularity).

6.2 Experimental Results

6.2.1 Latency

The latency measured as the RTT by the psapp application includes both basic communication latency and the latency contributed by virtualization. Virtualization introduces latency in two ways: (1) a packet must be classified as to which VIF it belongs to, and (2) the guest domain owning this VIF must be notified. Based on the MAC address of the packet and using hashing, classification can be done in constant time for any number of VIFs, assuming no hash collision.

For the HV-NIC, step (2) above requires sending a signal from the driver to the guest domain. This takes constant time, but with increasing CPU contention, additional end-to-end latency would be caused if a target guest were not immediately scheduled to process the signal. Thus, with an increasing number of VIFs, we would expect latency values to increase and exhibit larger variances. Finally, since the driver domain and guest domains are scheduled on different CPUs, sending a signal to a guest domain involves an IPI (inter-processor interrupt).

For the SV-NIC, step (2) above requires the hypervisor to virtualize the PCI interrupt and forward it as a signal to the guest domain, as described in Section 4.3. In case the host core responsible for interrupt virtualization is being shared by the target guest domain, sending this signal is done via a simple upcall, which is cheaper than performing an IPI. Given that all guest domains are scheduled on two CPUs, on the average, signal forwarding from one of these two cores provides the performance optimization 50% of the time. As with the HV-NIC case, if multiple guest domains are sharing a CPU, the target guest domain may not be scheduled right away to process the signal sent by the self-virtualized network interface. Thus, with an increasing number of VIFs, we would expect latency values to increase and exhibit larger variances.

Another source of latency in device-centric case is the total number of signals that need to be sent per packet. As mentioned earlier in Section 4.3.1, due to the limitations on interrupt identifier size, a single packet may require more than one guest domains to be signalled. In particular, the total number of domains

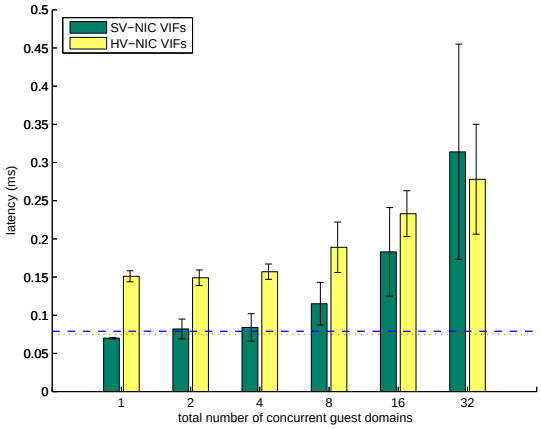


Figure 4: Latency of HV-NIC and SV-NIC. Dotted lines represent the latency for dom0 using the tunnel network interface in two cases: (1) No SV-IO functionality (*i.e.*, without software bridging), represented by fine dots, and (2) host-centric SV-IO functionality (*i.e.*, with software bridging), represented by dash dots

signalled, n_s, is given by the following formula: $\begin{cases} \lfloor n/l \rfloor \leq n_s \leq \lceil n/l \rceil & \text{if } n > l \\ 1 & \text{otherwise} \end{cases}$, where n is the total number of domains and l is the interrupt identifier size. Thus, the smaller the l, the more the number of domains that will be signalled (all but one of which would be redundant), and vice versa. Assuming these domains share the CPU, the overall latency will include the time it takes to send a signal and *possibly* the time spent for useless work performed by a redundant domain; latter of which will be decided by domain scheduling on the shared CPU.

Using RTT as the measure of end-to-end latency, Figure 4 shows the RTT reported by *psapp* for HV-NIC and SV-NIC. On the x-axis is the total number of concurrent guest domains 'n' running on 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 device-centric approach to SV-IO, it is possible to obtain close to a 50% latency reduction for VIFs compared to Xen's current host-centric implementation. This reduction results from the fact that dom0 is no longer involved in the network I/O path. In particular, the cost of scheduling dom0 to demultiplex the packet, using bridging code, and sending this packet to the front-end device driver of the appropriate guest domain is eliminated on the receive path. Further, the cost of scheduling dom0 to receive a packet from guest domain front-end and to determine the outgoing network device using bridging code is eliminated on the send path. Also, with SV-NIC, the latency of using one of its VIFs in a domU is almost identical to using the tunnel interface from the domain that has direct device access, dom0. Our conclusion is that the basic cost of the device-centric implementation is low. Also demonstrated by these measurements is that the cost of our SV-NIC implementation is fully contained in the device and the HV.

The median latency value and inter-quartile range increases in all cases as the number of guest domains (and hence the number of simultaneous flows) increases. This is because of increased CPU contention between guests. Also, due to interrupt identifier sharing, the latency of SV-NIC increases beyond that of HV-NIC for 32 VIFs. In that case, every identifier bit is shared among 4 VIFs, and hence, requires 1.5 redundant domain

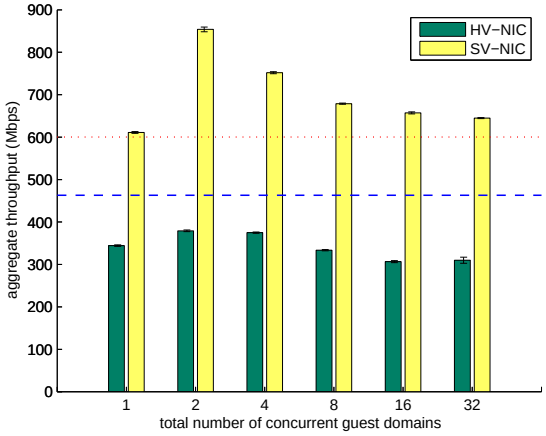


Figure 5: TCP througput of HV-NIC and SV-NIC. Dotted lines represent the throughput for dom0 using tunnel network interface in two cases: (1) No SV-IO functionality (*i.e.*, without software bridging), represented by fine dots, and (2) host-centric SV-IO functionality (*i.e.*, with software bridging),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

schedules on the average before a signal is received by the correct domain. On our system with only two CPUs available for domUs, these domain schedules also require context switching, which further increases latency.

Since latency degrades due to CPU contention among guests, we expect to see better performance if a large, *e.g.* a 32-way, SMP were used. In that case, SV-NIC would perform better for 32 guests even though the virtual interrupt identifier is shared. This is because all signalled domains would be running on different CPUs. The performance of the HV-NIC remains similar to the case of the single guest domain for a 2-way SMP (results for which are shown in Figure 4.)

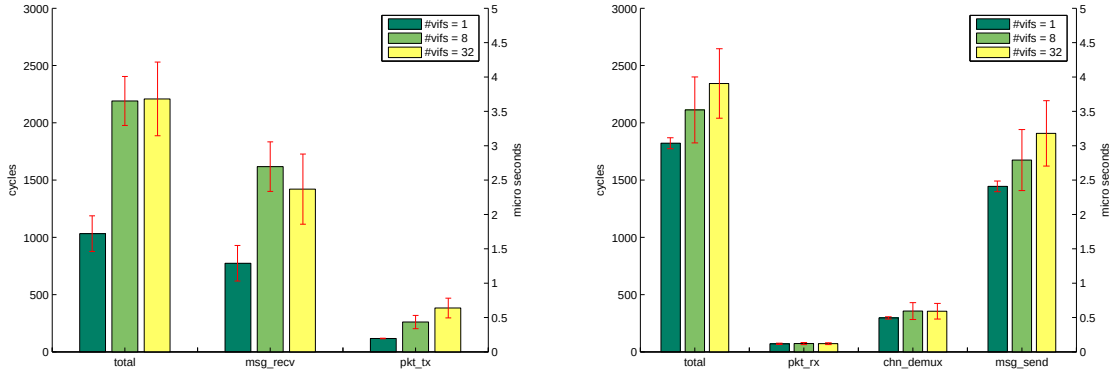
6.2.2 Throughput

The aggregate throughput achieved by n flows is the sum of their individual thoughputs. Particularly, if we denote the aggregate throughput for n flows as a random variable T, it will be equal to $\sum_{i=1}^{n} T_i$, where T_i denotes the random variable corresponding to the throughput for flow i. Since we expect each T_i to have finite mean μ_i and variance σ_i^2, and since they are independent of each other (they may not be normally distributed), we expect T to follow a normal distribution $N(\sum_{i=1}^{n} \mu_i, \sum_{i=1}^{n} \sigma_i^2)$ according to the central limit theorem. We estimate the mean and variance for each flow by the sample mean and variance.

Figure 5 shows the throughput of TCP flow(s) reported by iperf for SV-NIC and HV-NIC. The setup is similar to the latency experiment described above. The mean and standard deviation for the aggregate throughput of the 'n' simultaneous flows as computed above is shown on the y-axis.

Based on these results, we make following observations:

- The performance of the *HV-NIC is about 50% of that of SV-NIC*, even for large numbers of guest domains. Several factors contribute to the performance drop for the HV-NIC, as suggested in [26], including high L2-cache misses, instruction overheads in Xen due to remapping and page transfer between driver domain and guest domains, and instruction overheads in the driver domain due to software ethernet bridging code. The overhead for software bridging is significant, as demonstrated by the difference between the dotted lines in Figure 5. In comparison, the SV-NIC adds overhead in Xen for interrupt routing and for overhead incurred in the micro-engines for layer-2 software switching.
- The performance of using a single VIF in domU using the SV-NIC is similar to using the tunnel interface in dom0 without the SV-IO functionality. This shows that the cost of device-centric SV-IO realization is low and that it purely resides in the ENP2611 and the HV.



(a) Egress Path

(b) Ingress Path

Figure 6: Latency Microbenchmarks for SV-NIC

- 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 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).

6.3 SV-NIC Microbenchmarks

In order to better assess the costs associated with the SV-NIC, we microbenchmark specific parts of the micro-engine and host code to determine underlying latency and throughput limitations. We use cycle counting for performance monitoring on both micro-engines and the host.

6.3.1 Latency

Figures 6(a) and 6(b) show the latency results for the egress and ingress paths respectively on micro-engines.

The following sub-sections of the egress path are considered:

- *msg_recv* – The time it takes for the context specific to a VIF to acquire information about a new packet queued up by the host side driver for transmission. This involves polling the send queue in SDRAM.
- *pkt_tx* – Enqueueing the packet on the transmit queue of the physical port.

For the ingress path, we consider the following sub-sections:

- *pkt_rx* – Dequeueing the packet from the receive queue of the physical port.
- *channel_demux* – Demultiplexing the packet based on its destination MAC address.
- *msg_send* – Copying the packet into host memory and interrupting the host via PCI write transactions.

The time taken by network I/O micro-engine(s) for transmitting the packet on the physical link and for receiving the packet from the physical link is not shown, as we consider it part of network latency.

When increasing the number of VIFs, the cost of the egress path increases due to increased SDRAM polling contention by micro-engine contexts for message reception from the host. The cost of the ingress path does not show any significant change, since we use hashing to map the incoming packet to correct VIF receive queue. The overall effect of these cost increases on end-to-end latency is small.

For host side performance monitoring, we count the cycles used for message send (PCI write) and receive (local memory copy) by guest domain and for interrupt virtualization (physical interrupt handler, including

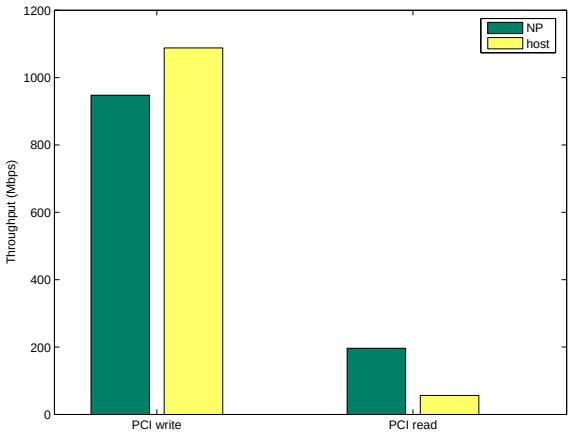


Figure 7: Throughput of the PCI interconnect between the host and the NP

dispatching the signals to appropriate guest domains) by Xen via the RDTSC instruction. For $\#vifs = 1$, the host takes $\sim 9.42\mu s$ for a message receive, $\sim 14.47\mu s$ for a message send, and $\sim 1.99\mu s$ for interrupt virtualization. For $\#vifs = 8$ and 32, the average cost of interrupt virtualization increases to $\sim 3.24\mu s$ and $\sim 11.57\mu s$, respectively, while the costs for message receive and send show little variation. The cost of interrupt virtualization increases since multiple domains might need to be signalled, even redundantly in the case when $\#vifs > 8$.

6.3.2 Throughput

We microbenchmark the available throughput of the PCI path between the host and the NP for read (write), by reading (writing) a large buffer across the PCI bus both from the host and from the NP. In order to model the behavior of SV-NIC packet processing, the read (write) was done 1500 bytes at a time. Also, aggregate throughput is computed for 8 contexts, where all of the contexts are copying data *without* any ordering requirement among them. This depicts the best case performance available for network I/O from NP to host. Results of this benchmark are presented in figure 7.

The results show the *asymmetric nature of the PCI interconnect, favoring writes over reads*. In addition, they demonstrate that the bottleneck currently lies in the ingress path of the SV-NIC. This is further exacerbated by ordering requirements and the need to use bounce buffers due to the limited addressability of host RAM from the NP. Better ingress path 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 link speed.

7 Architectural Considerations

7.1 Performance Impact of Virtual Interrupt Space

Currently, we only have a small (8 bit) identifier for interrupt source. Therefore, when the total number of VIFs exceeds the size of the identifier, an interrupt cannot be uniquely mapped as a signal to one VIF. This results in redundant signalling of guests domains and redundant checking for new network packets. Depending on the order in which a redundant signal is provided to domains, some domains might suffer cumulative context switching latency (when they cannot be scheduled simultaneously due to CPU contention). In order to demonstrate this effect, we artificially restrict the size of the identifier, ranging from 1 bit to 8 bits. This identifier space is then shared among 8 domains, each ID shared among $\lceil 8/l \rceil$ domains where l is the number of bits. We then perform a latency experiment (using the *psapp* application described above) between guests

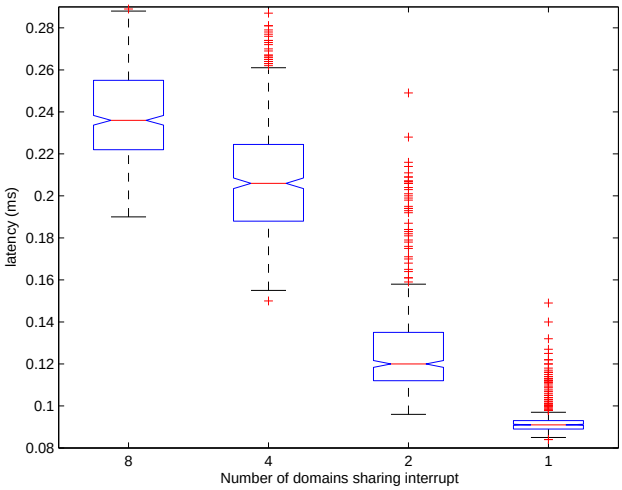


Figure 8: Effect of virtual interrupt sharing

that are assigned the 'last spot' in the sharing list for an ID. This setup explores the maximum latency before a signal will be delivered to the right domain. Figure 8 shows a boxplot of the median and inter-quartile range of the latencies. As expected, latency is reduced when domains do not share interrupt IDs. *These results advocate the use of large interrupt identifiers for device-centric SV-IO realizations.*

7.2 Insights for Future Multi-cores

In modern computer architectures, different interconnects (busses) connect system components like memory and CPU in a particular topology. These busses are themselves connected together via bridges, thereby providing a communication path between different components.

The result of organizations like these is non-uniform communication latency and bandwidth between different components; *e.g.*, the communication path between a CPU core and memory is usually much faster and of higher bandwidth than the path between the CPU core and I/O devices. While interconnect technologies have improved both throughput and latency for I/O devices, they still situate them relatively 'far' from processing units and subject their data streams to bus contention and arbitration through the chipset path. This is particularly problematic for devices with low-latency requirements or short data transfers that cannot take advantage of bursts. In particular, *it has a negative impact on device-centric SV-IO*, which may potentially need to issue a larger number of interactions in order to signal multiple domains housing their virtual devices.

In upcoming chip multi-processor systems (CMPs), multiple CPU cores are placed closely together on the same chip [6]. Furthermore these cores may be heterogeneous [11], to include specialized cores like graphics or math co-processors and network processing engines (similar to NPs), along with general purpose cores. These cores may also share certain resources, such as L2 cache, which can greatly reduce inter-core communication latency. Our results demonstrate that *a specialized multi-core environment will better support efficient realization of device-centric SV-IO.*

To quantify and compare the architectural latency effects of the current I/O data path in the multi-core paradigm, we run a simple 'ping-pong' benchmark that passes a short 32-bit message back and forth between (1) two distinct *physical* CPU cores, and (2) a CPU core and an attached NP using shared memory *mailboxes*. For the CPU-to-NP benchmark, the local mailbox for CPU core (NP) is present in host memory (NP's SDRAM), and the remote mailbox for CPU core (NP) is present in NP's SDRAM (host memory). For the CPU-to-CPU benchmark, all mailboxes are present in host memory.

We experiment with polling in both directions (Poll/Poll) *vs.* polling in one direction coupled with asynchronous notification (IRQ) in the other direction (Poll/IRQ), for receiving a message in the local

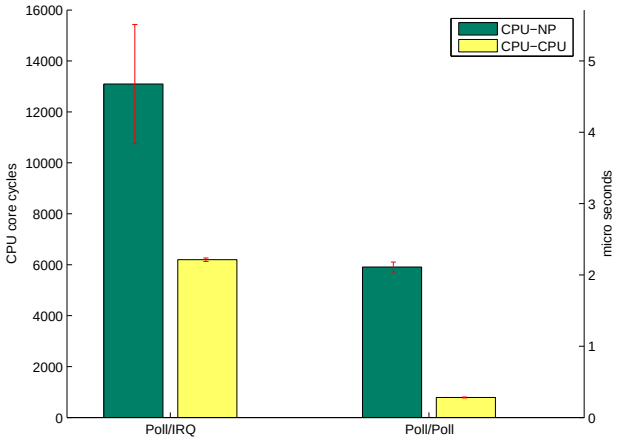


Figure 9: Communication latency for a simple ping-pong benchmark between two CPU cores and a core and an attached NP.

mailbox from the peer. The case of using IRQs in both directions is omitted, since the host CPU cannot send such a notification to the NP's packet processing cores (micro-engines.) Inter-processor interrupts (IPI) are used to send IRQ notifications between CPU cores, while a PCI interrupt is used to send the same from NP to host CPU.

Results are reported in Figure 9. Times are for a complete round-trip measured using the RDTSC instruction on host CPU. The difference between the core-to-core results and core-to-NP results is attributable to the difference in the length and complexity of the data path messages need to traverse. Since the NP is present as a PCI device, the path between the CPU cores, the NP, and their respective remote mailboxes is 'longer' than that between two CPU cores and their memory interconnect. This extra distance adds overhead as well as variance, especially in overload conditions when the various busses' scheduling and arbitration is under stress. The difference between polling and asynchronous IRQ notifications is caused by the costs of saving and restoring the CPU context and other OS-level interrupt processing as well as demultiplexing potentially shared IRQ lines.

From these results, it is clear that in the approaching multi-core world, there will be substantial benefits to be attained by (1) *positioning NP-like communication cores 'close' to the host computer cores*, and (2) *having many rather than few cores* so they can be dedicated to particular tasks and thus allow use of low-latency polling rather than slow and variable asynchronous interrupts. The communication latencies for polling-based solutions will be further reduced when these cores share resources like L2 cache, as that would reduce/eliminate the costs associated with cache invalidations and accessing system memory. The cost of interrupt-based solutions may also be reduced if the system bus connecting the interrupt controllers of all cores (*e.g.* LAPICs for x86 architecture) is also implemented on chip.

One tradeoff associated with spin-based polling is that it causes wasted CPU cycles, and therefore, additional energy consumption. It is possible to perform power efficient polling in recent processors via two new instructions, termed 'monitor' and 'mwait'. The CPU programs a dedicated memory snooping circuit via the monitor instruction, providing it the target memory location to be polled. It then enters a low power sleep state via the mwait instruction. The CPU is woken up in case of a write to the target memory location, or for other reasons (*e.g.*, external interrupts). A polling loop implemented using monitor/mwait attains latency comparable to spin based polling, albeit consumes less power. Since the older Xeon host cores in our testbed do not support these instructions, we quantify the latency of this approach using a Pentium extreme edition 3.2GHz processor-based machine with attached ENP2611 board. We then scale the cycle count numbers to the original testbed cores, assuming the absolute time for ping-pong benchmark would remain the same on both cores. Results are encouraging, as they show that *the benchmark takes* ~ 6536 *cycles* ($\sim 2.27\mu s$*), which is similar to the case when spin-based polling is employed.*

In-Place Data Manipulation

Specialized cores typically provide functionality that is costlier to implement on general purpose processors. For example, network processors perform network-specific processing on data, which can also include application-specific data processing, such as filtering information from message streams based on certain business rules [19]. In a multi-core system, the resultant data/information must also be made available to other cores in case they are executing other parts of application logic. Although there is resource sharing in modern systems to enable this (*e.g.*, the host CPU can access NP's SDRAM and vice-versa), *often the cost of accessing shared resources makes in-place data manipulation by different cores prohibitively costly* (as shown by our microbenchmarks, both NP and host PCI read bandwidth are very limited). As a result, multiple data copies must be made by different cores to their *local* memory before they can efficiently operate on it. This, coupled with the fact that application logic running at one core may not have complete knowledge of the information requirements of other cores, may result in large amounts of wasted memory bandwidth and increased latency due to redundant data copying.

Future multi-core systems will alleviate this problem, since all cores will be equidistant from main memory and hence will be able to access shared information at similar cost. However, the cost of accessing memory may still become a bottleneck in the case of collaborative in-place data manipulation by multiple cores. Early trends demonstrate that future multi-core systems will share caches at some level (*e.g.*, L2 cache) [6], but the increasing number of cores will raise multiple issues with cache sharing. First, in a multiple-CMP configuration where each chip has only a small number of cores, coherency will be an issue among caches on different chips, and will require more complex cache coherence protocols, such as Token coherence [24]. Second, for large-scale CMPs, with many cores on the same chip, large shared caches will no longer have a uniform access time, rather, that time will depend on the wire distance between the core and the specific part of the cache being accessed [21]. This might require restructuring applications' access behavior in order to extract good overall performance.

8 Related Work

Modern computer systems perform a variety of tasks, not all of which are suitable for execution on general purpose processors. A platform consisting of both general purpose and specialized processing capability can provide the high performance as required by specific applications. A prototype of such a platform is envisioned in [20] where a CPU and a NP are utilized in unison. Due to the limitations of the PCI interconnect, a special interconnect is designed that provides better bandwidth and latency for CPU-NP communication. More recently, multiple heterogeneous cores have been placed on the same chip [11]. These cores can share resources at a much lower level than shared memory (such as L2 cache), thus greatly improving both bandwidth and latency of inter-core communication. Our work uses a similar heterogeneous platform, consisting of Xeon CPUs and an IXP2400 NP communicating via a PCI interconnect. For this platform, performance advantages are demonstrated for a device-centric realization of SV-IO. This is in comparison to other solutions that use general purpose cores for network packet processing and other device-near tasks [30, 16].

The SV-IO abstraction bears resemblance to the virtual channel processor abstraction proposed in [25], although the intended use for virtual channel processors is to provide virtual devices for some system-level functionality, such as iSCSI, rather than guests having to run their own iSCSI module which in turn communicates to the virtual network interface. VIFs provided by SV-IO can be similarly enhanced by added functionality. As a concrete example, we are currently working to build privacy enhanced virtual camera devices. These *logical* VIFs provide a extension mechanism similar to *soft-devices* [36], major difference being that we can flexibly choose where to extend the VIF, whether at the device level directly or at the host level. This flexibility is provided by utilizing both host- and device-resident processing components.

In order to improve network performance for end user applications, multiple configurable and programmable network interfaces have been designed [39, 29]. These interfaces could also be used to implement a device-centric SV-NIC. Another network device that implements this functionality for the zSeries virtual-

ized environment is the OSA network interface [9]. This interface uses general purpose PowerPC cores for the 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 SV-IO implementation. Furthermore, these virtual interfaces can be efficiently enhanced to provide additional functionality, such as packet filtering and protocol offloading.

Our SV-NIC uses VMM-bypass in order to provide direct, multiplexed, yet isolated, network access to guest domains via VIFs. The philosophy is similar to U-Net [35] and VMMC [18], where network interfaces are provided to user space with OS-bypass. A guest domain can easily provide the VIF to user space applications, hence 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.

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 [15]. 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 [32, 19, 40].

9 Conclusions and Future Work

In this paper, we advocate the SV-IO abstraction for I/O virtualization. We also enumerate various design choices that can be considered for the realization of this abstraction on modern computer systems. Specifically, we present the design and an initial implementation of a device-centric SV-IO realization as a *self-virtualized network interface device* (SV-NIC) using an IXP2400 network processor-based board. Performance of the virtual interfaces provided by this realization is analyzed and compared to a host-centric SV-IO realization on platforms using the Xen hypervisor. The performance of device-centric SV-IO is better than that of the host-centric SV-IO. It also scales better with an increasing number of virtual interfaces used by an increasing number of guest domains.

Our SV-NIC enables high performance in part because of its ability to reduce HV involvement in device I/O. In our solution, the HV on the host is responsible for managing the virtual interfaces presented by the SV-IO, but once a virtual interface has been configured, most actions necessary for network I/O are carried out without HV involvement. Here, a limiting factor of our current hardware is that the HV remains responsible for routing the interrupt(s) generated by the SV-IO to appropriate guest domains. Future hardware enhancements, such as larger interrupt ID spaces and support for message signaled interrupts may alleviate this problem.

To improve upon the current performance of VIFs, we plan to make the following changes to our current implementation:

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

These changes will improve the performance for both device- and host-centric realizations of SV-IO.

As part of future work, we are investigating *logical* virtual devices built using SV-NICs. 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. These devices can also be used for *remote virtualization* of devices present on different hosts and thereby provide a virtual device level abstraction to guest domains. Finally, logical virtual devices could be enhanced with certain QoS attributes.

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 SV-IO abstraction was motivated by research interactions with Jun Nakajima at Intel. Finally, we acknowledge help and input from other students in our group, most notably, Sanjay Kumar and Sandip Agarwala.

References

[1] AMD I/O Virtualization Technology Specification. `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/34434.pdf`.

[2] ENP-2611 Data Sheet. `http://www.radisys.com/files/ENP-2611_07-1236-05_0504_datasheet.pdf`.

[3] Global environment for network innovations. `http://www.geni.net`.

[4] IBM eserver xSeries ServeRAID Technology. `ftp://ftp.software.ibm.com/pc/pccbbs/pc_servers_pdf/raidwppr.pdf`.

[5] Intel 21555 Non-transparent PCI-to-PCI Bridge. `http://www.intel.com/design/bridge/21555.htm`.

[6] Intel Pentium D Processor Specification. `http://www.intel.com/products/processor/pentium_D/index.htm`.

[7] Intel Virtualization Technology Specification for the IA-32 Intel Architecture. `ftp://download.intel.com/technology/computing/vptech/C97063-002.pdf`.

[8] Iperf. `http://dast.nlanr.net/projects/Iperf`.

[9] OSA-Express for IBM eserver zSeries and S/390. `www.ibm.com/servers/eserver/zseries/library/specsheets/pdf/g2219110.pdf`.

[10] Tcpdump/libpcap. `http://www.tcpdump.org/`.

[11] The Cell Architecture. `http://www.research.ibm.com/cell/`.

[12] The VMWare ESX Server. `http://www.vmware.com/products/esx/`.

[13] Intel IXP2400 Network Processor: Hardware Reference Manual, October 2003.

[14] D. Abramson et al. Intel Virtualization Technology for Directed I/O. *Intel Technology Journal*, 10(3), 2006.

[15] H. Bos, W. de Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis. Ffpf: Fairly fast packet filters. In *Proc. of OSDI*, 2004.

[16] T. Brecht et al. Evaluating Network Processing Efficiency with Processor Partitioning and Asynchronous I/O. In *Proc. of EuroSys*, 2006.

[17] B. Dragovic et al. Xen and the Art of Virtualization. In *Proc. of SOSP*, 2003.

[18] C. Dubnicki, A. Bilas, K. Li, and J. Philbin. Design and implementation of virtual memory-mapped communication on myrinet. In *Proc. of the International Parallel Processing Symposium*, 1997.

[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] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proc. of ASPLOS*, 2002.

[22] R. Kumar. *Holistic Design for Multi-core Architectures*. PhD thesis, Department of Computer Science and Engineering, 2006.

[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] M. R. Marty, J. D. Bingham, M. D. Hill, A. J. Hu, M. M. K. Martin, and D. A. Wood. Improving multiple-cmp systems using token coherence. In *Proc. of HPCA*, 2005.

[25] D. McAuley and R. Neugebauer. A case for Virtual Channel Processors. In *Proc. of the ACM SIGCOMM 2003 Workshops*, 2003.

[26] A. Menon et al. Diagnosing performance overheads in the xen virtual machine environment. In *Proc. of VEE*, 2005.
[27] F. Petrini, D. Kerbyson, and S. Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *Proc. of Supercomputing*, 2003.
[28] I. Pratt et al. Xen 3.0 and the Art of Virtualization. In *Proc. of the Ottawa Linux Symposium*, 2005.
[29] I. Pratt and K. Fraser. Arsenic: A User Accessible Gigabit Network Interface. In *Proc. of INFOCOM*, 2001.
[30] G. Regnier et al. ETA: Experience with an Intel Xeon Processor as a Packet Processing Engine. *IEEE Micro*, 24(1):24–31, 2004.
[31] E. Riedel and G. Gibson. Active disks - remote execution for network-attached storage. Technical Report CMU-CS-97-198, CMU, 1997.
[32] 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.
[33] J. H. Salim, R. Olsson, and A. Kuznetsov. Beyond softnet. In *Proc. of 5th USENIX Annual Linux Showcase and Conference*, 2001.
[34] V. Uhlig et al. Towards Scalable Multiprocessor Virtual Machines. In *Proc. of the Virtual Machine Research and Technology Symposium*, 2004.
[35] 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.
[36] A. Warfield, S. Hand, K. Fraser, and T. Deegan. Facilitating the development of soft devices. In *Proc. of USENIX ATC*, 2005.
[37] R. West et al. Dynamic window-constrained scheduling of real-time streams in media servers. *IEEE Transactions on Computers*, 2004.
[38] D. Wetherall, J. Guttag, and D. Tennenhouse. Ants: A toolkit for building and dynamically deploying network protocols. In *Proc. of IEEE OPENARCH*, April 1998.
[39] P. Willmann, H. Kim, S. Rixner, and V. Pai. An Efficient Programmable 10 Gigabit Ethernet Network Interface Card. In *Proc. of HPCA*, 2005.
[40] H. youb Kim and S. Rixner. Tcp offload through connection handoff. In *Proc. of EuroSys*, 2006.