Virtualization Services: Accelerated I/O Support in Multi-Core Systems

Himanshu Raj, and Karsten Schwan,
College of Computing,
Georgia Institute of Technology,
Atlanta, GA 30332

***Abstract*—Virtualization services permit I/O subsystems and peripheral devices to be virtualized by placing select functionality on specialized cores and/or on cores situated 'closer' to devices. The approach is used to implement *self-virtualized I/O* (SV-IO), which off-loads certain virtualization functionality onto the cores available on the I/O device, accelerating I/O functions, efficiently using key performance-limiting resources in multi-core systems, i.e., memory and I/O bandwidth, and exploiting the parallelism inherent in multi-core architectures. This paper evaluates a concrete instance of self-virtualized I/O, a self-virtualized network interface (SV-NIC), targeting the high end NICs used in datacenters. Experimental evaluations of the SV-NIC in a prototyping environment using an IXP2400-based ethernet board show high scalability in terms of the numbers of virtual interfaces (VIFs) offered to guests, and up to ∼77% improvements in throughput and ∼53% reductions in latency, compared to the current standard virtualized device implementations on hypervisor-based platforms. Beyond such performance advantages, the generality of virtualization services and their use for implementing enhancements to standard services are demonstrated with a storage service that provides location transparent access to block devices for guest VMs, and with enhancements to a network service that provide per-VM, priority-based servicing of virtual network devices.**

I. Introduction

Virtualization technologies are making it possible to map the concurrent computing workloads of multiple and often highly diverse applications to multi-core platforms that range from server systems to lower end desktops, laptops, and even handheld devices. For such platforms, virtualization must deal with the potential overheads both of using multiple cores and of platform heterogeneity in terms of performance differences in communication paths, e.g., inter-core communication channels, memory buses, and I/O buses as well as differences across computational cores in their speeds or capabilities or even their instruction set architectures (ISAs) [1], [2]. For instance, the overheads of scheduling multiple virtual machines may severely limit the performance of virtualized I/O [3]. Another basic question that virtualization solutions must address is the right level of abstraction - whether or not we should continue to virtualize multi-core platforms at the low levels of abstraction imposed by the dictate of 'hardware emulation'. Para-virtualization has already established that performance can be improved if a virtual device does not export to guest VMs the same API as its corresponding physical device [4], and researchers are exploring new approaches to organize the device interfaces used by guests [5], [6].

The research goal of this work is to better understand virtualization mechanisms and implementations on multi-core platforms. Focusing on I/O virtualization, we take a step in these directions that develops principles and associated software methods using which devices can be virtualized efficiently and in ways that can take advantage of the relative computational abilities of general purpose cores, of specialized cores, and of the diverse platform-level connectivities between cores and devices. Our approach uses notions from the domain of Service Oriented Architecture (SOA) to describe each resource exposed to guest VMs as a *virtualization service* (VS). Service consumers, which are guest virtual machines (VMs, also referred to as guest domains), use collections of such services to gain access to architectural resources like CPUs, memory, or I/O devices. More importantly, the services used by guests can extend beyond physical resources to provide new logical [7], [8] or enhanced physical resources, where enhancements may concern improved performance or platform scalability and/or additional value-added attributes and functionalities useful to VMs or required by underlying platforms. For example, a useful storage virtualization service is one that provides a virtual disk with additional reliability properties for the data stored on it, with costs commensurate with the degree of reliability offered.

The virtualization services approach advocated in this paper is guided by several implementation principles:

- *partitioning the computational cores* available on the platform across different services in ways that minimize the costs of scheduling and context switching when services are invoked;
- *minimizing inter-core communication*, to attain low latency for service execution by minimizing data copying and inter-core signaling;
- *specializing service implementation*, to better exploit the functionality of the underlying platform, e.g., network processing engines for processing network I/O; and
- *using service-specific abstractions*, that is, exploiting service semantics for gaining improved performance, rather than operating at the abstraction levels presented by physical devices.

These principles enable efficient virtualization services in multiple ways: (1) by creating per-resource, custom virtualiza-

H. Raj is now with Microsoft Research

tion solutions, including those that (2) exploit the considerable power of future off-the-shelf, many-core hardware platforms, and (3) by judiciously mapping service components to select computational cores. An example described in detail in this paper is a self-virtualized network interface (SV-NIC) that efficiently exploits specialized communication cores. Further, (4) virtualization services can go beyond simply virtualizing hardware to also providing semantically meaningful, novel functionality to guest VMs, an example being the enhanced storage service mentioned above. (5) Another interesting attribute of the virtualization services approach is that it can be used to deal with the hardware-imposed limits on memory and I/O bandwidths present in many-core machines. This is because due to its self-contained nature, each virtualization service's functionality can be mapped to platform resources so as to optimize bandwidth use, e.g., by reducing or removing the needs for data copying. Examples include the SV-NIC's use of bandwidth-conscious communication methods to communicate directly with VMs (i.e., using hypervisor-bypass methods), and a storage service built on top of the SV-NIC in ways that optimize data copying between the networking stack and storage stack.

This paper makes several novel contributions:

- It defines *virtualization service* as a fundamental construct for composing system-level virtualization solutions.
- Focusing on I/O, virtualization services offer a flexible implementation vehicle for I/O virtualization functionality. In particular, these services (i) encapsulate all of the tasks associated with virtualizing an I/O device, providing management APIs to the hypervisor (also referred to as the virtual machine monitor (VMM)) and virtual devices and associated access APIs to guest VMs, and (ii) offer flexibility in how I/O virtualization is implemented, allowing both *device-centric* realizations (also referred to as Self-Virtualized I/O, or SV-IO in brief) that use processing capabilities present *close* to the peripheral device as well as the more traditional *host-centric* realizations. In other words, a virtualization service constitutes an evolutionary approach to I/O virtualization – it encapsulates currently prevalent host-based I/O virtualization approaches, and it allows a more decentralized approach by using specialized cores and functional partitioning, such as those provided by future virtualization enhanced devices [9].
- The notion of virtualization services can be used to raise the level of abstraction at which platform resources are presented to guest VMs. I/O-centric examples are (i) a storage service offering transparent local vs. remote data storage, assisted by the SV-NIC, and (ii) a QoS-enhanced SV-NIC that provides varying degrees of behavior isolation among guest VMs via prioritization of virtual network devices. Such *logical devices* not only efficiently implement the data movements between virtual machines (VMs) and the virtualized platforms on which they run, but also capture semantic information about VM-device interactions that are then used to implement additional device functionality like that pertaining to the fair or efficient sharing of underlying physical devices.

The virtualization services approach is evaluated experimentally. An I/O service implementation for a gigabit network interface with on-board processing resources, demonstrates that a device-centric realization (SV-NIC) exhibits improved performance ($\sim 2X$ better throughput, higher scalability, and $\sim 50\%$ less latency) than a host-centric realization (also referred to as HV-NIC). Partitioning virtualization processing actions across multiple cores on a per service basis is also shown useful for interrupt virtualization, where dedicating a specific host core to this task along with using the SV-NIC implementation provides an up to 50% latency reduction for 32 guest VMs vs. the case when the interrupt virtualization task is shared by all cores. Other services able to benefit from partitioning include those providing page table updates or facilitating VM-VMM communication in VT-enabled [10] systems [11].

The remainder of this paper is organized as follows. Section II describes virtualization services for I/O as well as the *device-* and *host-centric* realizations. Section III presents the design and implementation of a network virtualization service for a high-end gigabit network interface, followed by a detailed description of the device-centric realization, termed SV-NIC, in Section IV. SV-NIC can further take advantage of the core partitioning 'sidecore' approach in a multi-core system, as described in Section V. Section VI presents new functionality provided by virtualization services, the example being a storage service that offers location transparent access to block devices for guest VMs. The section also describes interesting service enhancements, such as a network service that offers per-VM, priority-based servicing of virtual network devices. Both of these services are built on top of the SV-NIC. Detailed experimental evaluation of network virtualization service realizations is presented in Section VII, followed by related work in the areas of I/O virtualization and composing services in virtualized environments. Section IX concludes the paper with a summary of its contributions and explores future directions.

II. Virtualization Services: Toward Efficient Virtualized I/O

To enable efficient and high performance virtualized I/O, the virtualization services abstraction must describe the resources used to implement device virtualization. These resources include (1) some number of processing components (cores), (2) communication media connecting these cores to the physical device, and (3) the physical device(s) itself. With these resources, a *host-centric* service implementation, for instance, demands that all virtualization functionality run on host processing cores, including using a driver domain per device [12], using a driver domain per one set of devices [13], or running driver code as part of the hypervisor [4]. In contrast, alternative *device-centric* realizations, SV-IO, implement selected virtualization functionality on the device itself, resulting in less host involvement and potential performance or isolation benefits. With virtualization services, therefore, system developers have 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 [14]–[16], smart disk subsystems [17], [18], and even active network routers [19] with recent work focusing on network virtualization [20].

Functionally, a virtualization service (VS) for I/O must:

- multiplex/demultiplex a potentially large number of *virtual I/O devices* mapped to a set of physical devices in a scalable manner, Examples of virtual devices 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.
- provide a lightweight API to the hypervisor for managing virtual devices,
- efficiently interact with guest domains via simple APIs for accessing the virtual devices, and
- harness the compute power of many, potentially diverse processing cores.

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

A. Virtual Interfaces (VIFs)

At an abstract level, each VIF has 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);
void send(send queue, message m);
boolean isempty(receive queue);
message recv(receive queue);
```

Additionally, a pair of signals is associated with each queue for event-driven I/O. For the send queue, one signal is intended for use by the guest domain, to notify the VS that the guest has enqueued a message in the send queue. The other signal is used by the VS 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 VS are interchanged. A particular implementation of VS need not use all of these defined signals. For example, if the VS 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 VS. 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 [21] does not expect to receive any interrupts when it processes the receive operation for a bunch of incoming network packets.

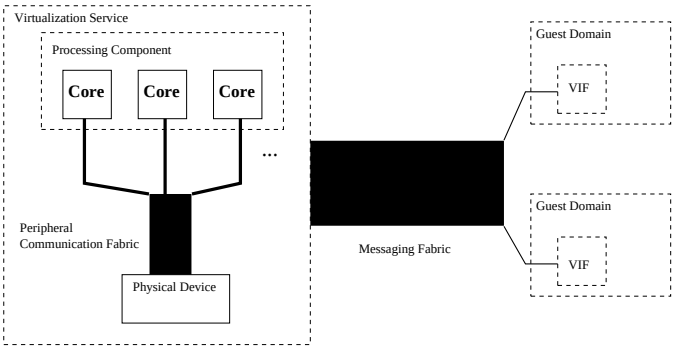


Fig. 1. Virtualization Service (VS) abstraction for I/O virtualization.

B. Virtualization Services: Design and Implementation

The VS 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 VS using VIFs via the *messaging fabric*.

The two main functions a virtualization service 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 VS'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 VS 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 [22], the simple scheduling method used in experimentation presented in this paper is round-robin scheduling.

The VS prototypes evaluated in this paper are implemented for the Xen hypervisor [4]. 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 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. The virtualization functionality provided by the driver domain for each physical device being virtualized is equivalent to that of a host-centric realization of a virtualization service. Host cores belonging to the driver domain are the VS's processing components, and they are architecturally homogeneous to the cores running guest domains. Host cores also run the VS 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.

The *device-centric* realization of a virtualization service, SV-IO, exploits the processing elements 'close to' the physical device [23]. In this case, the interconnect between the physical I/O device and on-device processing elements form the peripheral communication fabric, while the interconnect between the host system and the high end I/O device forms the messaging fabric, *e.g.*, PCIe. Performance and/or scalability for SV-IO are improved when it is possible to better exploit the device's processing resources, so as to improve device behavior due to 'fabric near' control actions [14], or to shorten the path from device to guest domain. A specific example of a SV-IO is presented in the next section.

Note that we use the terms device- or host-centric to refer to the location(s) of the majority rather than the entirety of the processing functionality in a VS. Our device-centric SV-IO implementation for network virtualization service, for instance, requires host assistance for certain control plane device/guest interactions. Similarly, host-centric realization will require some degree of device-level support, *e.g.*, the capability to perform I/O.

III. Network VS: Realizations for Network Interface Virtualization

A. Hardware Platform and Basic Concepts

The communication device used is an IXP2400 network processor(NP)-based RadiSys ENP2611 board [23]. 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) [24]. 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. This network device is attached to a x86-based host platform via PCI interconnect.

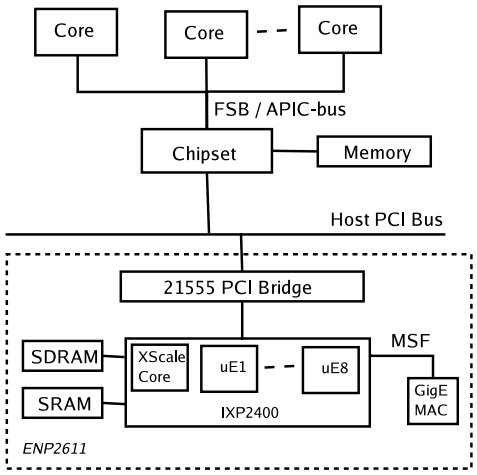


Fig. 2. Host-NP platform.

The combined host-NP platform represents one point in the design space of future multi-core systems, offering heterogeneous cores for running applications, guest OSes, and I/O functionality. As shown later, the platform is suitable for evaluating and experimenting with the scalability and with certain performance characteristics of the VS 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 [25]. 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.

Another notable feature of the host-NP platform is the limitation on PCI read bandwidth. Micro-benchmark results presented in Figure 3 demonstrate 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 network packet processing, the read (write) was done 1500 bytes at a time. Also, aggregate throughput for NP to host read/write is computed for a single micro-engine doing programmed I/O using all of its 8 available hardware contexts, where the contexts are copying data *without* any ordering requirement among them (as shown by results presented

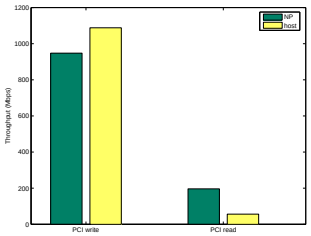


Fig. 3. Throughput of the PCI interconnect between the host and the NP.

elsewhere [26], ordering does not adversely impact the NP to PCI write throughput).

The results show the *asymmetric nature* of the PCI interconnect, favoring writes over reads. These results motivate our design choice for implementing message queues in NP SDRAM and host memory in a manner such that writes by a core are performed to the remote memory (across PCI bus), while reads by a core are performed to the local memory. This way, I/O is performed mostly via writes to PCI address space from both host and NP cores, thereby effectively utilizing the I/O communication bandwidth.

B. HV-NIC: Host-Centric Implementation of the Network VS

In the host-centric realization, the network interface's virtualization logic runs in the driver domain (or controller domain, Dom 0) 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. Figure 4 shows the architectural diagram of HV-NIC.

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 forwards packets queued by the device driver running in the driver domain to the network I/O logic, which sends it out to 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 received from the network I/O logic 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*

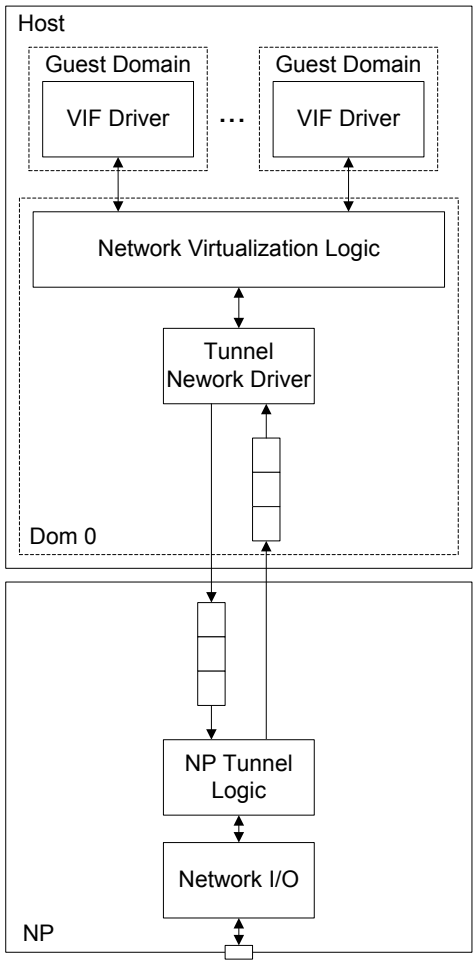


Fig. 4. HV-NIC: Host-centric network virtualization service for host-NP platform

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. For a detailed description of Xen's network interface virtualization, we refer the reader to [13].

C. SV-NIC: Device-Centric implementation of the Network VS

In our device-centric SV-IO implementation, termed SV-NIC, most of the processing component, the peripheral communication fabric, and the physical I/O device components of the VS abstraction are situated on the ENP2611 board itself. Specifically, 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 processing component situated on the device performs the key tasks of network I/O, and its multiplexing/de-multiplexing to various guest VMs, as depicted in Figure 5. The peripheral communication fabric consists of the *Media and Switch Fabric* (MSF) [24]. The physical I/O device, the PM3386 gigabit

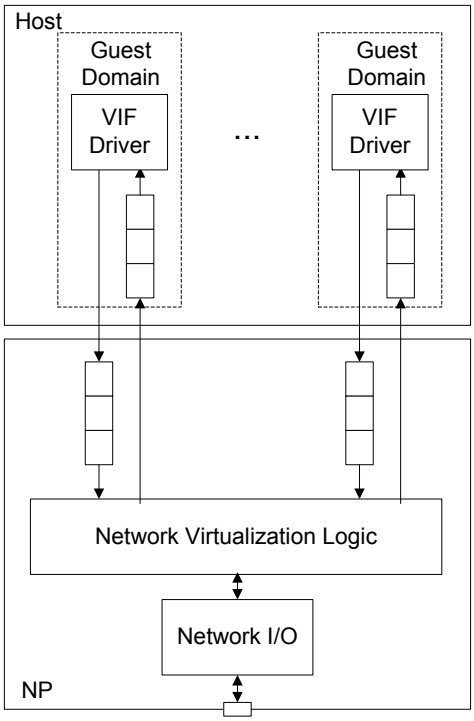


Fig. 5. SV-NIC: Device-centric network virtualization service for host-NP platform

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 devices. A detailed description of the functionality breakdown of various SV-NIC processing components is presented in the next Section.

IV. SV-NIC: Implementation Details

In this section, we describe how we map the design and implementation principles presented earlier to the SV-NIC realization. In particular, we utilize the principles of *partitioning* and *specialization* to use the NP cores for network I/O virtualization. The SV-NIC realization is also a *low latency design*, since it effectively uses the parallelism provided by multiple hardware contexts of NP cores and implements a fast path for I/O to guest VMs by providing them with safe, direct access to the network card.

A. Management

Management functionality includes the creation of VIFs, their removal, and changing the attributes and resources as-

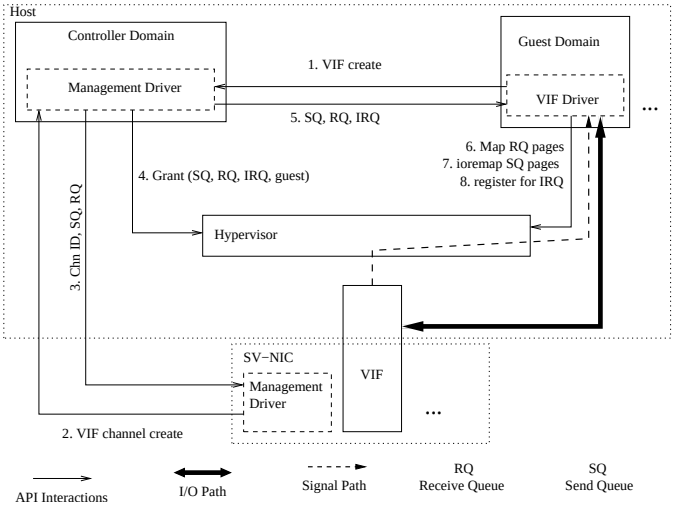


Fig. 6. Management interactions between SV-NIC, hypervisor and the guest domain to create a VIF. Shaded region depicts the boundary of SV-IO abstraction.

sociated with them. This functionality is realized via 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) and executes on the host core(s). The device-side driver is part of the embedded OS running on the NP-based board and It runs on the XScale core. Management requests are generated by the VIF driver in guest domains or the hypervisor itself. The interactions among VIF driver and host-side management driver are carried out over a *xenbus* channel, a shared memory, message-based, inter-VM communication mechanism in Xen. The hypervisor communicates with the host-side management driver using a shared memory buffer as well. All management related decisions are made by 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 6 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).

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. Throughput 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.

The hypervisor plays a key role of privilege management to enable safe direct access of SV-NIC resources to guest domains. In particular, for a guest domain to utilize the VIF provided by the SV-NIC for network I/O, 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 safely inject messages 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.

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 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 directly perform any management related functionality, it cannot influence the NP to perform any illegal I/O to a VIF that it does not own.

B. Network I/O

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.

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.

In our SV-NIC implementation, only some of the signals associated with VIFs are needed: those sent from the SV-NIC to the guest domain. In particular, the SV-NIC sends signals to guest domains related to status of send and receive queues, and its micro-engines poll these queues for information from host. 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 high latency. 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 VIF 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 NP cores process packets at a rate slower than the rate at which the host core(s) enqueue them). 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.

Signaling is achieved by assigning to every VIF 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. The implementation, uses a simple round robin assignment policy, where the identifier assigned to a guest VM, ID_{vm}, is computed as $ID_{vm} = ID{+}{+} \bmod 8$, where ID is set to 0 at SV-NIC initialization. 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 is implemented as part of the Xen hypervisor itself. In particular, The master PCI interrupt generated by the SV-NIC is sent to a specific host core that runs the signal demultiplexing and forwarding logic for 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.

In summary, the SV-NIC realization incorporates the VS design principle of *low latency* to exploit the parallelism inherent in NP cores and implements efficient VMM-bypass I/O. SV-NIC also employs *specialization* provided by the NP cores to better match networking related tasks to the capabilities provided by the platform. Coupling this with the principle of *partitioning*, SV-NIC provides an efficient and scalable network I/O virtualization solution that effectively utilizes the heterogeneous multi-core platform. In the next section, we provide another example of how SV-NIC employs core partitioning to support I/O virtualization.

V. Enhancing SV-NICs with Host-core Partitioning: the 'Sidecore' Approach

To generalize the core-partitioning approach, we next explore structuring a hypervisor itself as multiple components. These components are responsible for certain virtualization functionality, are internally implemented to best meet their obligations, and are mapped to different sets of cores. In multi- and many-core systems, components can even execute on cores other than those on which their functions are called. Furthermore, it becomes possible to 'specialize' cores, permitting them to efficiently execute certain subsets of rather than complete sets of hypervisor functionality. A similar componentization approach in multi-processor systems is taken by the K42 operating system [27].

We evaluate this implementation principle by experimentally dedicating a single core, termed *sidecore*, to perform specific hypervisor functions. This sidecore differs from *normal* cores in that it only executes one or a small set of hypervisor functionality, whereas normal cores execute generic guest VM and hypervisor code. A service request to any such sidecore is termed a *sidecall*, and such calls can be made from a guest VM or from a platform component, such as an I/O device. The result is a hypervisor that attains improved performance by internally using the client-server paradigm, where the hypervisor (server) executing on a different core performs a service requested by VMs or peripherals (clients). We demonstrate the viability and advantages of the sidecore approach by using it with the SV-NIC, in order to enhance the I/O virtualization capabilities via efficient interrupt virtualization. Other use cases are described in detail elsewhere [11].

As described earlier in Section IV, in the egress path, the micro-engines poll for packets in the guest VM's send-queue, which obviates the need of any involvement of the VMM or the driver domain. However, in the ingress path, the SV-NIC needs to signal the guest VM when packets are available for processing on the receive queue. Since the SV-NIC is a PCI device, it does so by generating a single master PCI interrupt, which is then routed to a host core by the I/O APIC. The master interrupt is intercepted by Xen, and based on the association between bits in the identifier register and VIFs, a signal is routed to the appropriate guest VM(s). Specifically, the master PCI interrupt is generated by the SV-NIC via a 8-bit wide identifier register – setting any bit in this register generates the master interrupt on the host. Hence, the SV-NIC can uniquely signal guest VMs for up to 8 VIFs. However, when the number of VIFs exceeds 8, these bits are shared by multiple VIFs, which may result in redundant signaling of guest VMs and may cause performance degradation.

In the sidecore approach, we use a host core to carry out the interrupt virtualization task. We establish a separate messaging channel between the micro-engines and the sidecore. This messaging channel is created in host-memory accessible via PCI I/O to micro-engines and local memory I/O to the sidecore. In the ingress path, micro-engines enqueue a message containing the ID of the VM that requires signaling. The sidecore continuously polls this messaging channel, and based on the ID contained in the message, sends a signal to the corresponding VM, signifying that one or more packets are available in the receive queue of the VIF. This approach not only improves performance, by avoiding the need for explicit interrupt routing, but it also improves performance isolation between multiple guest VMs. One example is a signal sent by SV-NIC for a VIF whose corresponding guest VM is not currently scheduled to run on the host core where the master interrupt is delivered. In our current implementation without sidecore, such a signal unnecessarily preempts the currently running guest VM to the hypervisor context for interrupt servicing. In contrast, the sidecore approach uses one host core exclusively for interrupt virtualization and hence, does not interrupt any guest VM unnecessarily. Further, we abandon signaling via PCI interrupts altogether, which reduces the latency of the signaling path by avoiding redundant signaling. A negative element of the approach is that all signals must always be forwarded by the sidecore 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 host core processing master interrupt to the guest domain in case the intended guest domain is currently scheduled to run on the same host core. Evaluation of SV-NIC

implementation with sidecore support is presented alongside SV-NIC without sidecore and HV-NIC in Section VII.

VI. Logical Devices: Enhanced I/O Virtualization Services

In virtualized environments, VM communication performance is an important aspect of overall system performance. Focusing on VM communications related to I/O and leveraging the fact that modern systems already have to virtualize the physical devices used by VMs, this section describes the performance advantages derived from extending virtual device interfaces, that is, from raising the level of abstraction offered to guest VMs by virtual devices by presenting them as *logical* instead of physical devices.

Constituting the fourth design principle of a virtualization service presented earlier, i.e., *service-specific abstractions*, logical devices take an information-centric view of how communication happens in the end systems. In this view, virtual machines executing on different CPU cores (and the applications they run) use the data sent and received for certain tasks and when doing so, extract or derive semantically meaningful information from such data. In fact, past work has already demonstrated many useful methods for accelerating such derivations by providing semantic information at lower levels, e.g., with active disks [28] or with enriched network interfaces [15]. Leveraging these methods, our approach is to permit end systems to associate with VMs' data communications the information extraction, annotation, and/or data conversion tasks they wish to have performed on said data. The goal of such associations is to improve certain end system characteristics, such as performance, scalability, and reliability, and to provide VMs with additional functionality at no or minimal cost. Performance improvements are derived from dealing with impediments in the critical paths of information exchanges. The aforementioned data conversions to correct for differences in endian-ness constitute one such impediment. Others include dealing with how I/O is handled in virtualized systems, e.g., by OS/VMM bypass, or how I/O resources are allocated among VMs to meet their QoS requirements. Examples of services related to scalability and reliability include online monitoring to help a virtual machine better manage a platform's resources, such as memory [29], and online monitoring to isolate misbehaving VMs.

Focusing on virtualized I/O, the platform associates some computation with a device I/O path. This association provides the VM with an enhanced virtual device, termed a *logical device*, with additional attributes/functionality which may not be natively supported by the corresponding physical device. Before continuing, we note that it is difficult to cleanly distinguish 'data' from 'information'. This is because the same unit may be treated as data by some software modules and as information by others. A concrete example is a unit of block device data handled by the device driver layer. It may contain file system specific directory information, i.e., information from the point of view of the file system, or data in a memory page from the operating system's point, or a structure implementing a binary search tree from an application's point of view. Virtualization services, therefore, leave it up to the end systems that handle these units to state and implement their information-centric views.

We leverage the virtualization services framework based on Xen virtualized environment presented earlier for building logical devices. For a host-centric virtualization service implementation, additional functionalities and properties required to implement logical devices are implemented inside the driver domain. Alternatively, using 'smart', self-virtualized devices, these functions are executed by the device itself. For the former host-centric virtualization service, the overall I/O path (and hence the latency) experienced by data to proceed from the physical device to the VM is longer, since the VP must schedule and run multiple VMs for each interaction. However, the cost of implementation is low due to the ease of programmability on standard x86 hardware. For the latter SV-IO based approach, the latency experienced by data movement is reduced, since data moves directly from device to guest VM, as described in Section IV. However, with more functionality desired from logical devices, the cost of building a solution with this approach may increase due to the increased complexity and cost of software development on a specialized platform. In this paper, we focus on logical devices for device-centric virtualization services. Realizations based on the host-centric approach are presented elsewhere [7], [30].

This section presents two concrete examples of enhanced virtualization services that provide logical devices: a network virtualization service that provides virtual NICs with priority-based QoS-support, and a storage virtualization service which permits a VM to access a block device regardless of whether such a device is physically located locally or must be accessed at a remote location. Xen-based implementations of these services demonstrate substantial performance improvements and additional functionality derived from the corresponding logical devices at a minimal cost to VMs, in part because virtualization services can utilize additional computational resources and can take better advantage of certain underlying platform capabilities.

A. QoS Enhanced Self-Virtualized NIC

Dynamic VM/application behaviors and consequent changes in the resource needs of their data flows require that virtualization services be aware of VMs' quality requirements. These requirements are either inferred implicitly by the VS [29], or are explicitly communicated by VMs. In this work, we chose the latter approach. In particular, for network I/O, we enhance the self-virtualized NIC (SV-NIC) prototype described earlier with priority-based QoS support, where flows from different VMs can be assigned different priorities. The management domain assigns a numeric priority between 1 and 100 to a VM and communicates this value to the SV-NIC via management driver (refer to Figure 6). The SV-NIC, then, uses the priority information to compute scheduling policies and resource allocation requirements for all VIFs corresponding to all VMs, where resources include network processor resources, such as IXP microengine contexts and interrupt identifier bits, and memory resources available on the SV-NIC. A more detailed

description of how the QoS feature is implemented on the NP board appears elsewhere [31].

B. SV-NIC based Remote Virtual Block Device (RVBD)

A simple logical functionality currently implemented by guest operating systems is that of a network block device (NBD) [32]. With NBD, block devices (e.g., disks) can be accessed remotely by having the guest operating system extract disk block information from the network packets it receives. Enhanced virtualization services enable an alternative approach that provide to guest VMs transparent remote device accesses. In this approach, a remote virtual block device (RVBD) provided by an enhanced *network VS* hides from the VM the fact that the physical device is located remotely.

The potential advantages of this approach are multi-fold. First, 'lean' functionality like that of RVBD can be implemented by the hardware, in a manner similar to iSCSI [33]. This is not likely the case for file system based realizations of remote data accesses. Second, there is already ongoing work that aims to decouple the efficient remote data accesses realized by approaches like RVBD from the complex semantics of modern file systems. An example is the Light-Weight File System (LWFS) created for the high performance domain [34] which separates fast path file read and write operations from operations used for meta-data purposes, such as file naming or consistency. The extended VS approach makes it easy to vary the placement of different elements of LWFS and/or its back-end storage functionality (e.g., object stores [35]) into and/or outside the virtualized platforms being used for their implementation. Third, decoupling the device access from device location significantly helps in device consolidation in large computing systems. Fourth, this approach removes the requirement that the guest VM runs the networking stack for disk access, thereby reducing the guest's computational resource needs. Finally, having transparent access facilitates *virtual device migration* [36] while doing VM migration, i.e., it provides continued access to I/O devices during and after the guest VM migration.

Before describing the RVBD implementation with SV-NIC, we briefly outline the Netbus mechanism [36], an extension of the basic shared memory based 'xenbus' communication mechanism used for virtual device access in systems virtualized with Xen. This mechanism utilizes a 'split' implementation of device driver stacks, consisting of a front-end (FE) and a back-end (BE) driver. The VM executes the FE, and the virtualization service executes the BE (either in the driver domain or in the self-virtualized device). FE and BE communicate with each other over Xenbus. To enable these communications to extend across multiple machines, Netbus extends Xen's existing single-platform solution by further splitting the BE into two components, local BE (LBE) and remote BE (RBE). With this approach, when a virtual device is added to a VM running on host M1 and if the corresponding physical device is remote (present on M2), the LBE on M1 establishes a communication channel with the RBE on M2. The LBE then tunnels data between the FE and the RBE. Both LBE and RBE constitute the virtualization service, along with any extensions required for logical functionality that can be implemented either in LBE or in RBE.

The RVBD implementation follows the Netbus approach described above, where the LBE implements the logical functionality. In particular, the RVBD FE is similar to that of a normal VBD FE. The RVBD FE inside the VM accesses the virtual disk device by making block requests to its corresponding RVBD LBE running at M1, which converts these requests to remote access requests and forwards them to the RVBD RBE, which runs at M2. The RVBD RBE then makes the actual requests to the device and returns the responses to the RVBD LBE over the network. The LBE performs the logical translation of these responses to VBD responses, and in turn returns the VBD information to the RVBD FE.

Previous work [36] has presented a host-centric virtualization services based RVBD solution with performance comparable to that of the NBD and with the added benefits described above. In particular, the LBE runs in Dom0 and communicates with the RBE on M2 over a TCP/IP connection. The RVBD FE is a regular VBD FE which is connected to the LBE. LBE converts the block requests and send them to RBE over the TCP/IP connection. The SV-NIC based solution evaluated in this paper takes the device-centric virtualization service approach, where the LBE is implemented on the NP board itself. Since a TCP offload solution is not available to us for the IXP-based NP card, we developed an alternative realization that uses message passing over ethernet to implement remote access. This 'lean' approach is in keeping with similar implementations done in the past [37] and with ongoing work in the high performance domain. In particular, the FE sends disk blocks as messages to the LBE on the NP card, which converts these messages to ethernet frames and sends them to the RBE. In the ingress path, the FE extracts disk blocks from messages received by the LBE and hands it over to the guest VM, without requiring data to traverse the guest VM's networking stack.

VII. Performance Evaluation

This section presents comparative experimental evaluation of the host-centric network virtualization service implementation (HV-NIC), along with the device-centric (SV-NIC) realizations with, and without, sidecore. It also presents evaluation of logical devices built using SV-NIC.

A. 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 gigabit ethernet card, which connects it to a separate subnet for developmental use.

Hosts are dual core hyper-threaded Pentium Xeon (a total of 4 logical processors) 2.80GHz servers, with 2GB RAM. The hypervisor used for system virtualization is Xen version 3.0 [13]. Dom0 runs a para-virtualized Linux 2.6.16 kernel with a RedHat Enterprise Linux 4 distribution, while guest VMs run a para-virtualized 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 guest VMs. Dom0 is configured with 512MB RAM, while each guest VM is configured with 32MB RAM. For the SV-NIC implementation enhanced with sidecore functionality, logical processor 0 is assigned to Dom0, while logical processor 1 is used as the sidecore (both of these belong to the same CPU core). For the SV-NIC without sidecore, both logical processors 0 and 1 are assigned to Dom0. The other two logical processors (2 and 3, belonging to the same CPU core) run the guest VMs. These logical processors share many architectural resources among them, such as caches and execution units. However, each logical processor has its own local APIC. Hence, an interrupt can be directed to a specific logical processor, without impeding the other one sharing resources with the target logical processor. The Borrowed Virtual Time (bvt) scheduler with default arguments is the domain scheduling policy used for Xen.

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).

B. SV-NIC vs. HV-NIC Performance Comparison

This section presents the results of two sets of experiments. The first set uses the 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 device-centric SV-NIC realization, both without and with the sidecore approach. The SV-NIC *vs.* HV-NIC realizations, *i.e.*, of their virtual interfaces provided to guest domains, are evaluated with two metrics: latency and throughput. These evaluations, presented next, provide evidence for the design and implementation principles advocated earlier for building high-performance and scalable virtualization services. In particular, experimental results demonstrate that there are significant benefits of *partitioning* network related functionality and moving such functionality to *specialized* network processing cores, along with judicial inter-core communication to achieve a *low-latency* design.

1) Latency: For latency, a simple libpcap [38] 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.

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 inter-processor interrupt (IPI).

For the SV-NIC without sidecore, 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 IV. 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 for SV-NIC without sidecore is the total number of signals that need to be sent per packet. As mentioned earlier in Section IV, due to the limitations on interrupt identifier size, a single packet may require more than one guest domains to be signaled. In particular, the total number of domains signaled, n_s, is given by the following formula:

$$\begin{array}{ll} \lfloor \frac{n}{l} \rfloor \leq n_s \leq \lceil \frac{n}{l} \rceil, & \text{if } n > l \\ 1 & \text{otherwise} \end{array}$$

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 signaled (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.

For the SV-NIC with sidecore, step (2) requires sidecore to receive a message from the NIC via a shared memory queue, and send a signal to the appropriate VM. Hence in the sidecore

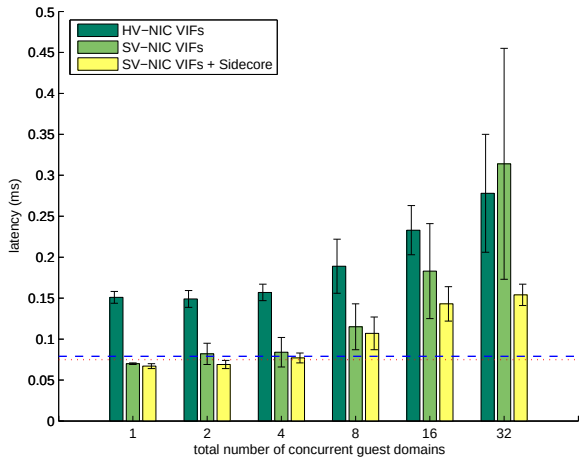


Fig. 7. Latency comparison of HV-NIC, SV-NIC, and SV-NIC with sidecore. Dotted lines represent the latency for Dom0 using the tunnel network interface in two cases: (1) No virtualization functionality (*i.e.*, without software bridging), represented by fine dots, and (2) host-centric network virtualization functionality (*i.e.*, with software bridging), represented by dash dots.

case, there is no PCI interrupt generated by the device and no redundant signaling of VMs.

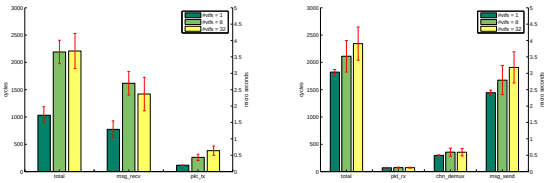
Using RTT as the measure of end-to-end latency, Figure 7 shows the RTT reported by *psapp* for HV-NIC and SV-NIC. On the x-axis is the total number of concurrent guest domains 'n' running on each host machine. On the y-axis is the *median* latency and inter-quartile range of the 'n' concurrent flows; 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 SV-IO approach, it is possible to obtain close to a 50% latency reduction for VIFs compared to a host-centric implementation.

Several factors contribute to the performance drop for the HV-NIC, as suggested in [3], including high L2-cache misses, instruction overheads in Xen due to remapping and page transfer between driver domain and guest domains, and instruction and scheduling 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 7.

In comparison, the SV-NIC adds overhead in Xen for interrupt routing on ingress path and for overhead incurred in the micro-engines for layer-2 software switching. In order to better assess the costs associated with the SV-NIC, we micro-benchmark specific parts of the micro-engine and host code to determine underlying latency limitations. Figures 8(a) and 8(b) show the latency results for the egress and ingress



(a) Egress Path (b) Ingress Path

Fig. 8. Latency micro-benchmarks for SV-NIC.

paths respectively on micro-engines for 64-byte packets.

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. Also, the *msg_send* cost in the ingress path for both without and with sidecore cases is similar, since both cases require a PCI bus transaction - to set a bit in the doorbell register for the former vs. to set a bit in host memory for the latter. The overall effect of these cost increases on end-to-end latency is small.

To demonstrate the impact of host-side functionality on overall SV-NIC latency, we measured the time for message send (PCI write) and receive (local memory copy) by guest domain for small packets, and for interrupt virtualization (physical interrupt handler, including dispatching the signals to appropriate guest domains) by Xen via the high-fidelity timestamp counter. 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, both with and without sidecore. 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, for SV-NIC without sidecore while the costs for message receive and send show little variation. The cost of interrupt virtualization increases since multiple domains might need to be signaled, even redundantly in the case when

$\#vifs > 8$. With sidecore, the cost of interrupt virtualization shows insignificant variation.

Overall latency reduction results following the principles of partitioning and specialization of cores. In this low latency design, 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. These networking specific tasks are performed by NP-cores. Also, with SV-NIC, the latency of using one of its VIFs in a guest VM is almost identical to using the tunnel interface from 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 hypervisor.

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 mostly because of increased CPU contention between guests. Also, due to interrupt identifier sharing, the latency of SV-NIC without sidecore optimization 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 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. In comparison, there is no redundant signaling for the SV-NIC with sidecore, which significantly improves latency. In terms of variability as depicted by the inter-quartile range, the sidecore approach provides less variability since the cost of signaling for every VM is similar - the sidecore always signals the core where the target VM is executing via an IPI.

2) Throughput: For throughput, we use the iperf [39] 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 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. The aggregate throughput achieved by n flows is the sum of their individual throughputs. Figure 9 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.

Benchmark results clearly demonstrate the benefits of using the device-centric SV-NIC approach over host-centric HV-NIC, for similar reasons described earlier. Overall, these results strongly advocate our implementation principles for building virtualization services, specifically *partitioning of cores*, and *minimizing inter-core communication* for cores that are situated farther apart. Results also demonstrate the effect sidecore has on throughput for smaller number of guest

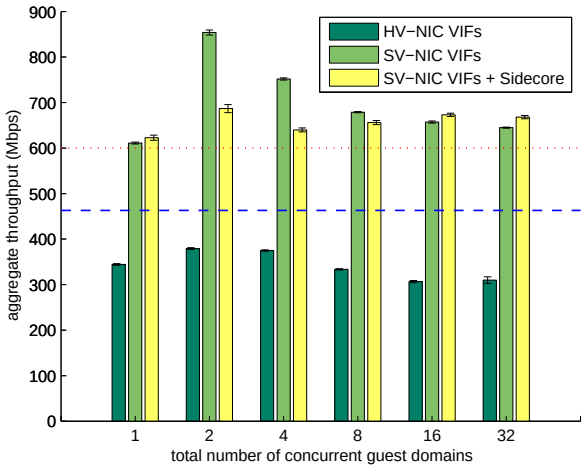


Fig. 9. TCP throughput comparison of HV-NIC, SV-NIC, and SV-NIC with sidecore. Dotted lines represent the throughput for Dom0 using tunnel network interface in two cases: (1) No virtualization functionality (*i.e.*, without software bridging), represented by fine dots, and (2) host-centric network virtualization functionality (*i.e.*, with software bridging), represented by dash dots.

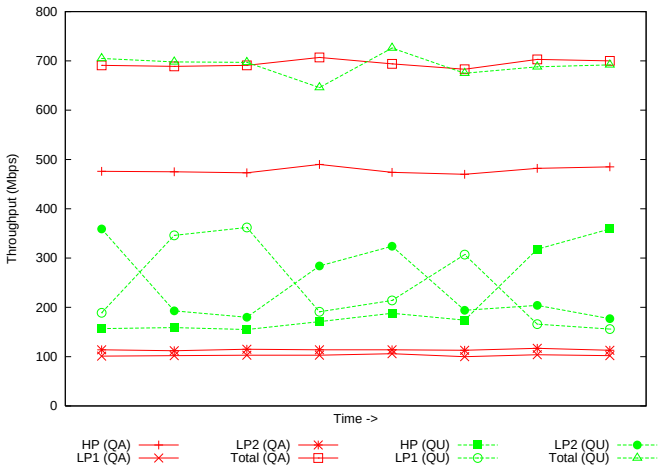


Fig. 10. Quality-Aware (QA) vs. Quality-Unaware (QU) SV-NIC.

VMs, since polling and synchronization operations reduce overall memory bus performance and negatively impact the I/O throughput for guest VMs. For larger number of guest VMs, this effect becomes secondary to the issue of core sharing among multiple VMs.

C. QoS-Enhanced SV-NIC

Figure 10 shows the benefits derived from the QoS-enhanced network virtualization service implemented by SV-NIC. The experimental setup is similar to that of the throughput experiment, except that three VMs, with one VIF each, are used to perform network I/O. One VM is set at high priority (HP), and the other two are set at lower priority (LP1 and LP2). The priority information is communicated to the service as described earlier. Here, the QoS-enhanced service, termed quality-aware (QA) service, correctly recognizes a situation of high input rate for all three flows and switches

TABLE I
LATENCY MICRO-BENCHMARK FOR PROVIDING A RESPONSE TO GUEST VM VIA RVBD AND NBD. IN BOTH CASES, SV-NIC PROVIDES EITHER A RVBD OR A VIF, RESPECTIVELY.

	Latency (ms)/response	Interpolated latency (ms)
RVBD	0.323	0.323
NBD	2.01	0.89

to providing priority servicing to the HP VM, which results in better throughput for this VM. In contrast, the quality-unaware (QU) service, lacks the functionality which results in all flows receiving a random percentage of the total egress throughput at any point in time.

D. SV-NIC based RVBD

Table I shows latency results for this low latency SV-NIC based RVBD solution and compares it to the NBD based solution. In particular, we measure the time taken by SV-NIC to provide a RVBD response to the guest VM's FE driver, which includes the time taken by the driver to copy the data from the bounce buffers of network I/O to buffer cache pages. This copy is required due to the limited host memory accessibility of our current IXP-based board. In contrast, for the VM-based implementation of this logical functionality, provided by NBD, the network packet is copied to the socket buffer from the bounce buffers by the network driver, and the rest is handled by the guest VM's networking stack. The application used for this benchmark is standard *hdparm* tool without prior caching of data, so as to measure the sequential read performance of the virtual disk. The average response size for NBD is 123570 bytes, while for RVBD, the average response size is 37601 bytes. We interpolate NBD's latency to a similar response size as that of RVBD. Results show that for response size of 37601 bytes, implementing logical block device functionality via RVBD provides a 64% latency reduction for a guest VM. The difference in latency is attributed to the removal of an extra copy (no need for movement via socket buffer to buffer cache) and the removal of networking stack. The SV-NIC implements the functionality by inspecting the packet header to identify the RVBD data, and handing it over to the logical block device driver running in guest VM. The cost incurred by the SV-NIC to implement this functionality is negligible ($\sim .3\mu s$) compared to the latency incurred at the host side.

VIII. RELATED WORK

This paper advocates the partitioning of a VMM's functionality when running on multi-core platforms. In particular, device-centric SV-IO provide I/O virtualization to guest VMs by utilizing the processing power of cores on the I/O device itself. The sidecore approach further enhances this approach to use partitioning when host core(s) carry out certain system virtualization tasks. A similar approach is used in operating systems, where processor partitioning is used for network processing [40], [41]. Computation spreading [42] attempts to run similar code fragments of different threads on the same core and dissimilar code fragments of the same thread on different cores. Another approach is to run hardware exceptions on a different hardware thread (or core) instead of running it on the same thread (core) [43]. While these solutions are targeted for better utilization of the micro-architecture resources such as caches, branch predictors, instruction pipeline etc., our solution is targeted at improving VMM performance and scalability for large scale many-core systems. Intel's McRT (many-core run time) [44] in 'sequestered' mode uses dedicated cores to run application services in non-virtualized systems. This approach requires major modifications in the application to utilize the parallel cores. This is in contrast to the sidecore approach, which requires only minor modifications to the guest VM's kernel and is aimed at improving the overall system performance.

Heterogeneous multicore systems provide the opportunity for task based componentization of modern computer systems, since not all of the tasks 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 by Hady et al [45] where a CPU and a NP are utilized in unison. 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 SV-IO realization of the network virtualization service. This is in comparison to other solutions that use general purpose cores for network packet processing and other device-near tasks [41], [46].

At a high level, our work uses a "multi-kernel" design akin to that of new operating systems for many-core architectures, like Barrelfish [47] and Helios [48]. However, the focus of our work is different, in that we build *virtualization services* by opportunistically using/modifying the available runtime (e.g., hypervisor based virtualized platform or runtime available on the NP board) - in contrast with both Barrelfish and Helios where the focus is on runtime itself. Also, in contrast to Barrelfish, our experimentation focuses on heterogeneous architectures, where virtualization services are designed to exploit diverse core capabilities for specific hypervisor functionality. Our service model is also different than Helios, where the runtime exports a uniform high-level API (a variant of .NET) to the service, agnostic of the architecture, and services are targeted to use that API. Our model of service development targets available runtime and/or architectures. The virtualization services abstraction also bears resemblance to the virtual channel processor abstraction proposed by McAuley et al [49].

In order to improve network performance for end user applications, multiple configurable and programmable network interfaces have been designed [50], [51]. These interfaces could also be used to implement a device-centric SV-NIC. Another network device that implements this functionality for the zVM virtualized environment is the OSA network interface [52]. This interface uses general purpose PowerPC cores for the purpose, in contrast to the NP cores used by our SV-NIC. Similarly, CDNA [53] uses a FPGA to implement the device-centric SV-NIC functionality. We believe that using specialized network processing cores provides performance

benefits for domain specific processing, thereby allowing more efficient and scalable network virtualization service 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 [54] and VMMC [55], 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 [56] and ethernet [57] devices offer functionality that is akin to the 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. Parallel to our work, PCI SIG developed I/O virtualization specifications for virtualization-capable I/O devices [9]. Although the SV-NIC currently does not adhere to these specifications, it incorporates similar abstractions. For example, our notion of virtual interface (VIF) is similar to PCI SIG's notion of virtual function (VF). The SV-NIC realization also focuses on virtualization services besides the aspect of sharing the I/O infrastructure. These virtualization services can implement diverse software-based *virtual functions* using logical devices on a programmable peripheral or general purpose CPUs as described in this paper. A similar mechanism for building flexible virtualization services is also advocated by LeVasseur et al [6].

Past research has made multiple attempts at providing semantically enhanced devices/interconnects in order to provide useful functionality to applications. For example, semantic-disks associate filesystem level information with the disk drive [28] to provide better performance and functionality to operating systems. The logical devices approach makes a similar argument, albeit in a virtualized system, where guest VMs' interactions are semantically enhanced. This approach is similar to Xen's 'soft' devices [58]. However, our implementation also utilizes the underlying platform's capabilities for supporting efficient I/O in virtualized systems, such as self-virtualized devices, and it could take advantage of other computational resources such as accelerators [59].

The semantic information considered in specific instances of logical devices described in this work is explicitly shared by guest VMs with the virtualized platform. This is not necessary though, since it is possible to implicitly infer limited amounts of information by monitoring a guest VM's behavior. For example, a VMM can infer when memory pages are added and removed from the guest OS page cache [29]. This implicit inference allows building VMM-level services such as a working-set size estimator and better secondary cache management. Similarly, it is possible to infer application level communication topology in a VM-based grid environment [60]. This topology information can be used to adapt the environment itself, e.g., via VM migration or via communication overlay adaptation, to provide performance benefits to applications running inside guest VMs.

IX. Conclusions and Future Work

New design and implementation principles are needed to exploit future heterogeneous multi- and many-core platforms. Targeting the specific topic of I/O virtualization, this paper develops and experiments with the abstraction of virtualization services. Using a virtualization service as a building block, we are able to efficiently use the parallelism provided by multiple computational cores and judiciously utilize memory and I/O bandwidth to provide improved performance to guest virtual machines. Virtualization services can also be used to leverage semantic information available from guests, to provide them with enhanced functionalities. These enhancements are services that generalize or specialize certain platform capabilities, as well as those that offer entirely new, software-realized functions. We expect the task-based componentization methods implemented with virtualization services to become increasingly important in future heterogeneous multi- and many-core systems.

Implementations of a network virtualization service serve to demonstrate the viability and advantages of the virtualization service concept, in terms of improved performance, scalability, and by providing entirely new functionality to guest VMs. A specific example is 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 service realization on platforms using the Xen hypervisor. Experimental results show substantial improvements in performance (upto $\sim 2X$ better throughput and $\sim 50\%$ less latency) and scalability for device-centric SV-NIC. *The SV-NIC enables high performance in part because of its ability to reduce HV and driver-domain involvement in device I/O*. In our solution, the HV and the driver domain on the host are responsible for managing the virtual interfaces presented by the virtualization service, but once a virtual interface has been configured, most actions necessary for network I/O are carried out without HV and driver-domain involvement.

Performance improvements for both HV-NIC and SV-NIC may be possible by further application of implementation principles. Specifically, we can improve upstream throughput by replacing engine programmed I/O with DMA to improve upstream throughput. Further, by utilizing a NP-platform with near-identical PCI read and write performance [61], we can also replace programmed I/O performed by host cores with DMA. This will reduce the host CPU utilization. Similarly, a NP-platform with message signaled interrupts support and large number of messages per devices (such as in MSI-X) will alleviate the performance overhead due to interrupt ID sharing for large number of VIFs. Architectural support for I/O virtualization, such as virtualization aware I/O MMUs and interrupt remapping support [10] will further improve the performance and scalability of the SV-NIC. In particular, by integrating the SV-NIC with an I/O MMU to create a separate *device domain per VIF* [10], the cost of host CPU utilization for software I/O MMU implementation and interrupt virtualization functionalities will be reduced.

An interesting attribute of virtualization services is that they can also be used to functionally partition the hypervisor and I/O functions present in multi-core platforms. This creates an interesting future direction of research in which one can consider dynamic core partitions based on current guest VM behavior and available platform resources. This would entail replacing our current static mapping of hypervisor components to certain multi-core resources with dynamic mappings, possible based on current platform state (e.g., whether or not certain cores are in certain states, including idle vs. active states). Although the number of cores available in a system are expected to increase, a virtualization service should be enhanced to handle dynamic changes in architectural resources, in particular to the number and types of computational cores available to it. In this manner, the utility of the overall system can be optimized within changing platform or application environments. For example, if the guest VMs in a system do not fully utilize computational cores, these cores could be utilized by the virtualization services to provide enhanced functionality to these VMs. However, if the demand for computational cores from guest VMs increases, the system should be able to reposition some of these computational resources away from the virtualization service. Such dynamic repositioning of resources should not disable a virtualization service – it should continue functioning, albeit with possible performance degradation. Such dynamic re-assignment of cores will depend on the degree of heterogeneity in the system. For example, in case of cores with different ISAs and/or with non-uniform addressability to system memory, such as Cell- [1], NP-based platforms [23], and GPUs [62], a virtualization service might be limited to execute on a specific subset of cores [63]. On the other hand, cores with minor differences in ISAs [64] and with uniform addressability to memory (although there might be performance issues in case of NUMA organization), will provide a larger set of cores to the service – although the selection of cores may result in reduced/slower functionality provided by the virtualization service. Such architectural heterogeneity will also impact the task placement of a virtualization service. For example, simpler functionality like sidecore may be assigned to smaller cores without impacting the performance and possibly with a side-benefit of energy efficiency.

REFERENCES

[1] "The Cell Architecture," http://www.research.ibm.com/cell/, accessed February, 2006.

[2] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai, "The impact of performance asymmetry in emerging multicore architectures," in *Proc. of the International Symposium on Computer Architecture*, 2005.

[3] A. Menon, J. R. Santos, Y. Turner, J. Janakiraman, and W. Zwaenepoel, "Diagnosing Performance Overheads in the Xen Virtual Machine Environment," in *Proc. of VEE*, 2005.

[4] P. Barham *et al.*, "Xen and the Art of Virtualization," in *Proc. of SOSP*, 2003.

[5] J. Satran, L. Shalev, M. Ben-Yehuda, and Z. Machulsky, "Scalable i/o-a well-architected way to do scalable, secure and virtualized i/o," in *Proc. of Workshop on I/O Virtualization (WIOV)*, 2008.

[6] J. LeVasseur, R. Panayappan, E. Skoglund, C. du Toit, L. Lynch, A. Ward, D. Rao, R. Neugebauer, and D. McAuley, "Standardized but flexible i/o for self-virtualizing devices," in *Proc. of Workshop on I/O Virtualization (WIOV)*, 2008.

[7] H. Raj, B. Seshasayee, and K. Schwan, "VMedia: Enhanced Multimedia Services in Virtualized Systems," in *Proc. of MMCN*, 2008.

[8] B. Seshasayee, N. Narasimhan, A. Bijlani, A. Pai, and K. Schwan, "Vstore: efficiently storing virtualized state across mobile devices," in *MobiVirt '08: Proceedings of the First Workshop on Virtualization in Mobile Computing*, 2008.

[9] "PCI Express IO Virtualization and IO Sharing Specification," http://www.pcisig.com/members/downloads/specifications/pciexpress/specification/draft/IOV_spec_draft_0.3-051013.pdf, accessed October, 2006.

[10] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert, "Intel Virtualization Technology for Directed I/O," *Intel Technology Journal*, vol. 10, no. 3, 2006.

[11] S. Kumar, H. Raj, K. Schwan, and I. Ganev, "Re-architecting VMMs for Multicore Systems: The Sidecore Approach," in *Proc. of WIOSCA*, 2007.

[12] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski, "Towards Scalable Multiprocessor Virtual Machines," in *Proc. of the Virtual Machine Research and Technology Symposium*, 2004.

[13] I. Pratt, K. Fraser, S. Hand, C. Limpack, A. Warfield, D. Magenheimer, J. Nakajima, and A. Mallick, "Xen 3.0 and the Art of Virtualization," in *Proc. of the Ottawa Linux Symposium*, 2005.

[14] 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.

[15] A. Gavrilovska, K. Mackenzie, K. Schwan, and A. McDonald, "Stream Handlers: Application-Specific Message Services on Attached Network Processors," in *Proc. of the Symposium on High Performance Interconnects (HotI)*, 2002.

[16] H. youb Kim and S. Rixner, "TCP Offload through Connection Handoff," in *Proc. of EuroSys*, 2006.

[17] E. Riedel and G. Gibson, "Active Disks - Remote Execution for Network-Attached Storage," CS, Carnegie Mellon University, Tech. Rep. CMU-CS-97-198, 1997, www.pdl.cmu.edu/PDL-FTP/NASD/CMU-CS-97-198.pdf, accessed October, 2007.

[18] "IBM eserver xSeries ServeRAID Technology," ftp://ftp.software.ibm.com/pc/pccbbs/pc_servers_pdf/raidwppr.pdf, accessed October, 2005.

[19] D. J. Wetherall, J. Guttag, and D. L. Tennenhouse, "ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols," in *Proc. of IEEE OPENARCH*, April 1998.

[20] "Global Environment for Network Innovations," http://www.geni.net, accessed October, 2007.

[21] J. H. Salim, R. Olsson, and A. Kuznetsov, "Beyond Softnet," in *Proc. of 5th USENIX Annual Linux Showcase and Conference*, 2001.

[22] R. West, Y. Zhang, K. Schwan, and C. Poellabauer, "Dynamic Window-Constrained Scheduling of Real-Time Streams in Media Servers," *IEEE Transactions on Computers*, 2004.

[23] "ENP-2611 Data Sheet," http://www.radisys.com/files/ENP-2611_07-1236-05_0504_datasheet.pdf, accessed October, 2005.

[24] Intel Corporation, "Intel IXP2400 Network Processor: Hardware Reference Manual," October 2003.

[25] "Intel 21555 Non-transparent PCI-to-PCI Bridge," http://www.intel.com/design/bridge/21555.htm, accessed May, 2005.

[26] H. Raj, "Virtualization Services: Scalable Methods for Virtualizing Multicore Systems," Ph.D. dissertation, College of Computing, Georgia Institute of Technology, 2008.

[27] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. D. Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig, "K42: Building a Complete Operating System," in *Proc. of EuroSys*, 2006.

[28] M. Sivathanu, V. Prabhakaran, F. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Semantically-Smart Disk Systems," in *Proc. of FAST*, 2003.

[29] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Geiger: Monitoring the Buffer Cache in a Virtual Machine Environment," *SIGARCH Comput. Archit. News*, vol. 34, no. 5, 2006.

[30] H. Raj and K. Schwan, "O2S2: Enhanced Object-based Virtualized Storage," *Operating Systems Review*, October 2008.

[31] R. Niranjan, A. Gavrilovska, and K. Schwan, "Towards IQ-Appliances: Quality-awareness in Information Virtualization," CERCS, Georgia Institute of Technology, Tech. Rep. GIT-CERCS-07-06, 2007.

[32] "Network Block Device," http://nbd.sourceforge.net, accessed April, 2006.

[33] "Alacritech SES1800 Series iSCSI Scalable Network Accelerators for Volume Servers," http://www.alacritech.com/Products/Storage/SES1800.aspx, accessed February, 2007.

[34] R. A. Oldfield, A. B. Maccabe, S. Arunagiri, T. Kordenbrock, R. Riesen, L. Ward, and P. Widener, "Lightweight I/O for Scientific Applications," in *Proc. of Cluster Computing*, 2006.

[35] G. A. Gibson, D. P. Nagle, K. Amiri, F. W. Chang, E. Feinberg, H. G. C. Lee, B. Ozceri, E. Riedel, and D. Rochberg, "A Case for Network-Attached Secure Disks," CS, Carnegie Mellon University, Tech. Rep. CMU-CS-96-142, 1996.

[36] S. Kumar, S. Agarwala, and K. Schwan, "Netbus: A Transparent Mechanism for Remote Device Access in Virtualized Systems," CERCS, Georgia Tech, Tech. Rep. GIT-CERCS-07-08, 2007, http://www.cercs.gatech.edu/tech-reports/tr2007/git-cercs-07-08.pdf, accessed October, 2007.

[37] C. A. Thekkath and H. M. Levy, "Limits to low-latency communication on high-speed networks," *ACM Trans. Comput. Syst.*, vol. 11, no. 2, 1993.

[38] "Tcpdump/Libpcap," http://www.tcpdump.org/, accessed October, 2005.

[39] "Iperf," http://dast.nlanr.net/projects/Iperf, accessed October, 2005.

[40] M. Rangarajan, A. Bohra, K. Banerjee, E. V. Carrera, R. Bianchini, L. Iftode, and W. Zwaenepoel, "TCP Servers: Offloading TCP/IP Processing in Internet Servers. Design, Implementation, and Performance," Department of Computer Science, Rutgers University, Tech. Rep. DCS-TR-481, 2002.

[41] T. Brecht, J. Janakiraman, B. Lynn, V. Saletore, and Y. Turner, "Evaluating Network Processing Efficiency with Processor Partitioning and Asynchronous I/O," in *Proc. of EuroSys*, 2006.

[42] K. Chakraborty, P. M. Wells, and G. S. Sohi, "Computation Spreading: Employing Hardware Migration to Specialize CMP cores on-the-fly," in *Proc. of ASPLOS*, 2006.

[43] C. B. Zilles, J. S. Emer, and G. S. Sohi, "The Use of Multithreading for Exception Handling," in *Proc. of MICRO*, 1999.

[44] B. Saha, A.-R. Adl-Tabatabai, A. Ghuloum, M. Rajagopalan, R. L. Hudson, L. Petersen, V. Menon, B. Murphy, T. Shpeisman, E. Sprangle, A. Rohillah, D. Carmean, and J. Fang, "Enabling Scalability and Performance in a Large Scale CMP Environment," in *Proc. of EuroSys*, 2007.

[45] F. Hady, T. Bock, M. Cabot, J. Chu, J. Meinechke, K. Oliver, and W. Talarek, "Platform Level Support for High Throughput Edge Applications: The Twin Cities Prototype," *IEEE Network*, vol. 17, no. 4, July/August 2003.

[46] G. Regnier, D. Minturn, G. McAlpine, V. A. Saletore, and A. Foong, "ETA: Experience with an Intel Xeon Processor as a Packet Processing Engine," *IEEE Micro*, vol. 24, no. 1, pp. 24–31, 2004.

[47] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: a new os architecture for scalable multicore systems," in *Proc. of SOSP*, 2009.

[48] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt, "Helios: heterogeneous multiprocessing with satellite kernels," in *Proc. of SOSP*, 2009.

[49] D. McAuley and R. Neugebauer, "A Case for Virtual Channel Processors," in *NICELI '03: Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence*, 2003.

[50] P. Willmann, H. Kim, S. Rixner, and V. Pai, "An Efficient Programmable 10 Gigabit Ethernet Network Interface Card," in *Proc. of HPCA*, 2005.

[51] I. Pratt and K. Fraser, "Arsenic: A User Accessible Gigabit Network Interface," in *Proc. of INFOCOM*, 2001.

[52] "OSA-Express for IBM eserver zSeries and S/390," www.ibm.com/servers/eserver/zseries/library/specsheets/pdf/g2219110.pdf, accessed October, 2005.

[53] P. Willmann, J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox, and W. Zwaenepoel, "Concurrent Direct Network Access for Virtual Machine Monitors," in *Proc. of HPCA*, 2007.

[54] 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.

[55] 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.

[56] 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.

[57] "Intel Virtualization Technology for Connectivity," http://softwarecommunity.intel.com/isn/downloads/virtualization/pdfs/20137_LAD_VTc_Tech_Brief_r04.pdf, accessed October, 2007.

[58] A. Warfield, S. Hand, K. Fraser, and T. Deegan, "Facilitating the Development of Soft Devices," in *Proc. of USENIX ATC*, 2005.

[59] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: using data parallelism to program GPUs for general-purpose uses," in *Proc. of ASPLOS*, 2006.

[60] A. I. Sundararaj, A. Gupta, and P. A. Dinda, "Increasing Application Performance in Virtual Environments through Run-time Inference and Adaptation," in *Proc. of HPDC*, 2005.

[61] "NFE-i8000 Network Acceleration Card," Information available from http://netronome.com/web/guest/products/acceleration_cards, accessed October, 2007.

[62] "The Cell Architecture," en.wikipedia.org/wiki/Graphics_processing_unit, accessed July, 2009.

[63] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan, "Gvim: Gpu-accelerated virtual machines," in *HPCVirt '09: Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, 2009.

[64] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn, "Using os observations to improve performance in multicore systems," *Micro, IEEE*, vol. 28, no. 3, 2008.