RICE UNIVERSITY

New Architectures and Mechanisms for the Network Subsystem in Virtualized Servers

by

Kaushik Kumar Ram

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Alan L Cox

Alan L. Cox, Chair
Associate Professor of Computer Science
and Electrical and Computer Engineering

Scott Rixner

Scott Rixner
Associate Professor of Computer Science
and Electrical and Computer Engineering

Peter Varman

Peter J. Varman
Professor in Electrical and Computer
Engineering

Houston, Texas

November, 2012

ABSTRACT

New Architectures and Mechanisms
for the Network Subsystem in Virtualized Servers

by

Kaushik Kumar Ram

Machine virtualization has become a cornerstone of modern datacenters. It enables server consolidation as a means to reduce costs and increase efficiencies. The communication endpoints within the datacenter are now virtual machines (VMs), not physical servers. Consequently, the datacenter network now *extends into the server* and *last hop switching* occurs inside the server. Today, thanks to increasing core counts on processors, server VM densities are on the rise. This trend is placing enormous pressure on the network I/O subsystem and the last hop virtual switch to support efficient communication—both internal and external to the server. But the current state-of-the-art solutions fall short of these requirements. This thesis presents new architectures and mechanisms for the network subsystem in virtualized servers to build efficient virtualization platforms.

Specifically, there are three primary contributions in this thesis. First, it presents a new mechanism to reduce memory sharing overheads in driver domain-based I/O architectures. The key idea is to enable a guest operating system to reuse its I/O buffers that are shared with a driver domain. Second, it describes *Hyper-Switch*, a highly streamlined, efficient, and scalable software-based virtual switching architecture, specifically for hypervisors that support driver domains. The Hyper-Switch combines the best of the existing architectures by hosting the device drivers in a driver domain to isolate any faults and placing the virtual switch in the hypervisor to perform efficient packet switching. Further, the Hyper-Switch implements several optimizations—such as virtual machine state-aware batching, preemptive copying, and dynamic offloading of packet processing to idle CPU cores—to enable efficient packet processing, better utilization of the available CPU resources, and higher concurrency. This architecture eliminates the memory sharing overheads associated with driver domains. Third, this thesis proposes an alternate virtual switching architecture, called *sNICh*, which explores the idea of server/switch integration. The sNICh is a combined network interface card (NIC) and datacenter switching accelerator. This takes the Hyper-Switch architecture one step further. It offloads the data plane of the switch to the network device, eliminating driver domains entirely.

Acknowledgments

Over the course of my PhD, several people have played a direct or indirect role in helping me cross the finish line. I have no doubt in my mind that without them, I would not have been successful in this endeavor. Unfortunately, it is not possible to name every one of them here, for there were too many. However, I would like to take this opportunity to thank a few among them.

First and foremost, I thank my advisor, Prof. Alan Cox, for his support, guidance, motivation, and inspiration during my stay at Rice University. His insights, suggestions, and feedback have shaped this thesis in many ways. I am grateful for the opportunity to work with Alan. I thank Prof. Scott Rixner for co-advising and helping me with all my research and publications. It was never easy facing Scott during the practise talks, proposals, or defenses! Nevertheless, it was a great learning experience. I thank Prof. Peter Varman for being a part of the thesis committee and for providing feedback on the thesis.

I also had the opportunity to collaborate with several researchers from HP Labs—including Jose Renato Santos, Yoshio Turner, Jayaram Mudigonda, and Partha Ranganathan—over many projects. I learned a lot from all of them. I also gained valuable experience when I worked with them as an intern at HP Labs.

I am grateful to have been a part of the Rice Computer Systems group that included several outstanding students and faculty members. Specifically, I thank Brent Stephens, Jeff Shafer, Mike Foss, and Thomas Barr for their help, advise, and feedback on many occasions. I thank everyone in the Rice Computer Science department for making it an amazing place to study and work.

I had the pleasure of being surrounded by wonderful friends who were with me during the good times and helped me through some of the difficult times at Rice. I cannot thank them all enough.

Finally, I sincerely thank my parents for always being there for me, for supporting me, and for trusting me do to the right thing. I thank my sister for her love and support. I am excited for the new beginnings in my life, both personally and professionally.

Contents

Illustrations

Tables

Chapter 1

Introduction

Machine virtualization has become a cornerstone of modern datacenters. It enables server consolidation as a means to reduce costs and increase efficiencies [1]. The cloud-based service infrastructures, such as Amazon EC2 [2], Rackspace Cloud [3], Windows Azure [4], and numerous others, use machine virtualization as one of their fundamental building blocks. Further, it is also being used to support the utility computing model where users can "rent" time in a large-scale datacenter [2]. These benefits of machine virtualization are now widely recognized. Consequently, the number of virtual servers in production is increasing rapidly. Specifically, the research firm IDC announced in December 2010 that more than 23% of all servers and more than 70% of all workloads on new machines will be virtualized [5].

Machine virtualization has led to considerable changes to the datacenter network and the I/O subsystem within virtualized servers. In particular, the communication endpoints within the datacenter are now virtual machines (VMs), not physical servers. Consequently, the datacenter network now *extends into the server* and *last hop switching* occurs within the physical server. Further, the server's I/O subsystem has to virtualize the devices to safely share them among the VMs hosted on the server. This thesis focuses on architectures and mechanisms for the network subsystem in virtualized servers to support efficient network communication.

The machine virtualization technology has come a long way since it was first used in IBM mainframes in the 1970s. Today, thanks to advances in both software and hardware, processor and memory virtualization have been heavily optimized to provide excellent performance. Unfortunately, the virtualization of devices, especially network devices, has lagged behind. Further, until recently, there has not been a lot of interest in the industry to explore new I/O virtualization architectures. This was primarily because the existing solutions, despite their drawbacks, were deemed good enough for most machine virtualization deployments. But, today, this situation is changing, primarily due to the ever increasing processor core counts on servers and the resulting increase in server VM densities.

In modern datacenters, the amount of inter-server communication is already significant and is expected to rise further [6, 7]. As virtualization spreads throughout the datacenter, and as more and more VMs are hosted within a server, this trend is expected to translate into significant inter-VM communication. So the network I/O subsystem and the last hop virtual switch within the server need to be able to keep up to support efficient communication—both internal and external to the server. Otherwise, the network will be a performance bottleneck, and this will in turn significantly impact the server VM densities. Further, in a virtualized datacenter, a physical server is likely to be shared by several applications, across multiple customers. So the mechanisms that aid in isolating network traffic must now operate all the way to the VMs through the last hop switch.

The existing support for I/O device virtualization comes in many forms. While these solutions have improved over the years, a single solution has not emerged as the best. Instead, all the solutions have their pros and cons. For instance, pure software approach to support I/O virtualization is commonly used in many virtualization platforms. This is due in part to the rich set of features—including security, isolation, and mobility—that software-based solutions offer. These solutions can be broadly classified into driver domain and hypervisor-based I/O models based on how the devices are virtualized in software.

The driver domain I/O model provides a safe execution environment for physical device drivers by hosting them in a dedicated VM[1] (a driver domain). This I/O model is supported by hypervisors like Xen [8] and Hyper-V [9]. Alternatively, hypervisors like VMware's ESX server [10] and Linux KVM [11], locate their device drivers within the hypervisor. This inflates the size of the trusted computing base (TCB) and therefore, reduces the reliability of the system.

While the driver domain model offers several benefits, it also incurs significant CPU overheads. One of the major sources of overhead is the mechanism that is used to share memory between VMs. The memory sharing mechanism is needed to allow the driver domain to access the network packets in a guest VM's memory, to perform I/O on its behalf. For instance, in Xen, the memory sharing overheads account for nearly 60% of the CPU cycles consumed in the driver domain while processing network packets. The hypervisor-based model avoids these overheads since the hypervisor has direct access to all guest VMs' memory. So, currently, depending on which software I/O model is chosen, one can achieve either higher fault isolation and system reliability or higher performance but not both.

Yet another point in this design space is the direct I/O model that has been proposed as a way to eliminate most the software overheads associated with I/O virtualization and thus, to close the gap with native I/O performance [12–16]. However, direct I/O solutions sacrifice device transparency since they require device-specific code within the guest VM. Also, it also does not offer the same flexibility as software-based solutions to support features like VM migration. Despite these drawbacks,

[1]The terms domains and virtual machines are used interchangeably in this thesis.

direct I/O solutions are used in some virtualization platforms for their high performance.

There have also been several proposals for implementing the last hop virtual switch in virtualized servers. Typically, the virtual switch is implemented in software within the hypervisor or the driver domain [17, 18]. In fact, Cisco and VMware have also made this last hop switch look and behave similarly to other switches in the datacenter [19]. However, there are significant software overheads inherent in this approach that make it an inefficient solution. The primary problems involve the more advanced features of a datacenter switch, including packet processing—*e.g.*, access control list (ACL) matching—and security operations—*e.g.*, DHCP and ARP.

There have been efforts, primarily from industry, in developing alternate solutions that leverage the functionalities in existing switches. The fundamental idea entails routing all traffic, even traffic among VMs co-located on the same physical server, to an external switch [20, 21]. So the external switch also performs all the last hop switch-related packet processing. While this solution eliminates most of the software overheads, it inherently wastes network link bandwidth between the server and the external switch. A middle ground in the design space is to switch packets within the server's network interface cards (NICs). The NICs that implement the direct I/O model, also implement a last hop switch. However, they only implement rudimentary switching functionalities due to their limited packet processing capabilities.

This thesis presents a spectrum of I/O virtualization solutions for the network subsystem in virtualized servers to build efficient virtualization platforms. Specifically, it proposes architectures and mechanisms that overcome the drawbacks in the existing I/O device virtualization and last hop switching solutions. The proposed solutions span both hardware and software, and illustrate the various bottlenecks and trade-offs inherent in I/O virtualization. The first two contributions of the thesis show that it is feasible to achieve high performance without sacrificing system reliability or fault isolation when using pure software solutions. The third contribution of this thesis shows that it is feasible to implement a NIC-based solution that supports all datacenter switching functionalities. Finally, this thesis concludes with a vision for the future of networking in virtualized systems that is based on the contributions presented in this thesis.

1.1 Contributions

A New Memory Sharing Mechanism

First, this thesis presents a new memory sharing mechanism that is designed specifically for driver domain-based I/O architectures. The new mechanism significantly reduces the overheads incurred when memory is shared between the driver domain and the guest VMs during I/O operations. The key idea is to enable a guest operating system to reuse the shared I/O buffers across multiple I/O operations. This is achieved by taking advantage of temporal and/or spatial locality in a guest VM's use of I/O buffers. The new mechanism makes it simple for a guest OS to implement a reuse scheme. Specifically, it allows the guest OS to unilaterally revoke access to the shared I/O buffers at any time. Another benefit of the new mechanism is that it provides a unified interface for memory sharing, whether between guest VMs and driver domains, or between guest VMs and I/O devices using the IOMMU hardware. The new mechanism was evaluated in the Xen virtualized platform using Linux VMs, where it reduced the CPU cost during I/O operations by up to 45% and increased the throughput by up to 150%.

The Hyper-Switch Architecture

Second, this thesis introduces *Hyper-Switch*, a highly streamlined, efficient, and scalable software-based last hop virtual switching architecture, specifically for hypervisors that support driver domains. While the first contribution reduces the memory sharing overheads, the Hyper-Switch architecture eliminates them entirely. Traditionally, in virtualization platforms that use driver domains, the last hop switch is implemented inside the driver domain along with the device drivers. Instead, the key idea in proposed architecture is to move the virtual switch from the driver domain to the hypervisor. In particular, the hypervisor implements a fast, efficient data plane of a flow-based software switch while the driver domain continues to safely host just the device drivers.

Further, this thesis also presents several optimizations that enable high performance. This includes virtual machine state-aware batching of packets to mitigate the cost of hypervisor entries and guest notifications. Preemptive copying and immediate notification of blocked guest VMs to reduce packet processing latency. Further, whenever possible, the network packet processing is dynamically offloaded to idle CPU cores in the system. These optimizations enable efficient packet processing, better utilization of the available CPU resources, and higher concurrency. As a result, the proposed architecture enables much improved and scalable inter-VM network performance, while still maintaining the fault isolation property of driver domains. A Hyper-Switch prototype was implemented in the Xen virtualization platform. The Hyper-Switch architecture was evaluated using this prototype where it outperformed Xen's default network I/O architecture and KVM's vhost-net architecture. For instance, in the pairwise scalability experiments the Hyper-Switch achieved a peak net throughput of $\sim$81 Gbps as compared to only $\sim$31 Gbps and $\sim$47 Gbps under Xen and KVM respectively.

The sNICh Architecture

Third, this thesis proposes an alternate last hop switching architecture called *sNICh*, which explores the idea of server/switch integration. In this architecture, the proposal in the second contribution is taken one step further by offloading the data plane of the switch to the network device and thereby, eliminating the need for driver domains entirely. As the name implies, the sNICh is a combined NIC and datacenter switching accelerator. But using a hardware-only approach it is not feasible to incorporate advanced switching functionalities in a NIC without making it expensive and/or limiting its scalability. The sNICh solution overcomes these limitations to implement a full-fledged switch while enabling a low cost NIC solution, by exploiting its tight integration with the server internals. This makes sNICh more valuable than simply a combination of a network interface and a datacenter switch.

The sNICh architecture diverges from a conventional switch-on-the-NIC architecture in three ways. First, it separates the control and data planes in the last-hop switch. Whereas the data plane is implemented in hardware within the NIC, the control plane is implemented in host software. Second, it supports flow-based packet switching to ensure that the software path is not traversed on every packet. Finally, it takes advantage of DMA engines on the host-side of the I/O bus to avoid wasting I/O bus bandwidth. The sNICh architecture was evaluated using a software prototype where the sNICh hardware was emulated in software. The sNICh prototype outperformed and scaled better than the existing solutions.

1.2 Organization

This thesis is organized as follows:

- Chapter 2 provides background information on the current state-of-the-art solutions for networking in virtualized datacenters. It presents existing solutions for virtualizing I/O devices and for last hop switching in virtualized servers.
- Chapter 3 describes the first contribution of this thesis. It presents the existing memory sharing mechanism, its drawbacks, and finally, the new mechanism. It also provides a comprehensive evaluation of the new mechanism.
- Chapter 4 describes the Hyper-Switch architecture, the second contribution of this thesis. It presents a detailed description of the new architecture, some details of its prototype implementation, and a complete evaluation of the proposed architecture.
- Chapter 5 introduces the sNICh, the third contribution of this thesis. It explains the sNICh architecture in detail. It also provides an evaluation of this architecture using software emulation.
- Chapter 6 discusses prior research that closely relate to the contributions of this thesis.
- Finally, Chapter 7 offers concluding remarks and directions for future research.

Chapter 2

Background

This chapter provides background information on the current state-of-the-art solutions for networking in virtualized datacenters. It is organized as follows. Section 2.1 explains the challenges due to the increasing adoption of virtualization in datacenters. Section 2.2 presents the current solutions for safely virtualizing and sharing I/O devices. Finally, Section 2.3 describes the existing solutions for last hop switching in virtualized servers.

2.1 Datacenter Networking Challenges

The datacenter is becoming one of the most critical components of the modern computing infrastructure. This trend has manifested in several ways. Primarily, data intensive applications, such as Google's search engine, can only operate in large scale datacenters. However, even smaller applications—workplace applications, such as document editors and spreadsheets, are migrating to the datacenter as a part of the *cloud* environment. Further, the *utility computing* model is emerging, whereby it is cost efficient to "rent" time in a large scale datacenter, enabling clients to quickly scale up or down the amount of computing resources at their disposal.

To efficiently serve this ever increasing number and diversity of applications and customers, datacenters must address two inefficiencies: server sprawl and multiple poorly utilized networks.

Historically, physical servers were rarely shared across multiple clients, and in many cases not even across application instances of the same client, so that the necessary performance SLAs and the inter-customer isolation can be achieved. Typically, these servers are under-utilized and wasteful of power [22].

Most datacenters also contain several parallel networks: a traditional Ethernet, a Fibre Channel network for storage traffic, and an InfiniBand fabric to support cluster traffic. These parallel networks are not cost-effective for several reasons; they cost more to build, require multiple administrators, complicate cabling, and waste rack space and energy.

Virtualization offers a promising avenue towards reducing server sprawl, particularly when combined with many-core processors. Modern virtualization systems allow several servers to be effectively consolidated onto a single physical machine. Similarly, advances in Ethernet networking offer a promising avenue towards increasing network utilization in the datacenter. The rapid rise of Ethernet network link bandwidths combined with the advent of sophisticated switch-based mechanisms—such as VLANs, ACLs, and link schedulers— for safely multiplexing different clients and traffic types can facilitate fabric consolidation.

The networking subsystems of virtualized servers, however, present a major impediment for both server and fabric consolidation. Datacenter networks and the server I/O subsystems are both architected in a way that expects the physical server to be an *end-point* in the network, and do not efficiently support a virtualized server, which is in reality a *network* in itself of virtual machines (VMs). Lack of efficient support for switching and for mechanisms that aid in ensuring isolation (such as ACLs, VLANs and QoS) causes the following three major problems in datacenter networks:

1. Lack of efficient switching support within the server can affect server densities in

the near-future for two reasons. First, in most modern datacenters, inter-server communication is already significant, and is expected to increase further [6, 7]. For instance, in Amazon's EC2 utility datacenter, a request from an external client machine can make as many as 100 different servers exchange messages among them [23]. Second, the increasing core counts on processor chips can easily be utilized by co-locating the servers of such applications on the same physical machine. However, this will not be possible if the networking subsystem cannot keep up to provide efficient inter-VM packet switching.

2. Lack of efficient access control within a server complicates fabric consolidation. The central problem in fabric consolidation is to isolate different clients and traffic types from hurting each other when forced to share a common switch or link. For instance, a buggy (or malicious) client should not be allowed to direct its storage traffic to another client's parallel program VMs, leading to serious packet loss in their synchronization traffic. To fully ensure such isolation, one must enforce the access restrictions (and QoS guarantees) on *all* hops of an end-to-end path. However, in virtualized servers, the real end-points are the VMs, and hence the end-to-end path extends through the server, involving the I/O subsystem in the last-hop. If the I/O subsystem does not extend the isolation and instead, allows traffic from different clients and different types to interact, it renders the isolation enforced in the greater datacenter network completely useless and makes fabric consolidation impossible.

3. Lack of consistent management primitives that work over the entire end-to-end path substantially complicates management. Today, most data center administration is spread over two main organizations within the IT department: the

server admin group and the network admin group. In traditional datacenters, for the most part, these groups are able to work independently of each other. In a virtualized datacenter, however, the roles of these admin groups get intertwined leading to much manual configuration and inter-group communication. For instance, the server admin that previously never had to deal with the details of the network must now configure the I/O subsystem to extend the isolation (and QoS) between different traffic types and clients. Further, to make this configuration effective the server admin must understand how the network itself is handling this separation (for instance which VLAN with which set of ACLs is being used in the network for one client vs another). This today requires close manual co-operation between the administrators. Even today's automated management systems are not designed to handle this blurred line between server and network administration.

The network subsystem in a virtualized server has two major components. The first component is the I/O virtualization architecture that is used to share the I/O devices among the VMs. The second component is the last hop virtual switching architecture used to switch network packets within the server. Both these components have to be taken into consideration to solve the datacenter networking problems. However, the management problem is not discussed in this thesis.

2.2 I/O Virtualization Architectures

The I/O virtualization architectures used in virtualized servers can be broadly classified into three models. While these models are discussed in the context of network devices, they are applicable to all I/O devices.

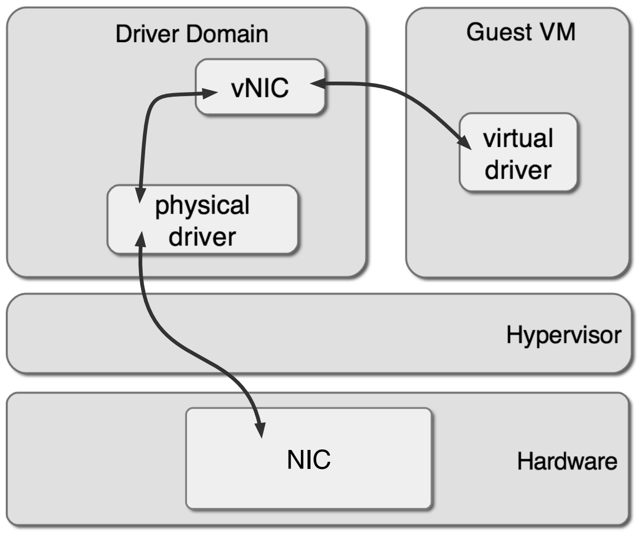


Figure 2.1 : *Driver domain I/O model. The driver domain, a dedicated VM, implements the virtual devices and also hosts the physical device drivers.*

2.2.1 Driver Domain I/O Model

In this I/O model, each guest VM is provided a virtual network interface (vNIC), which is implemented completely in software within a dedicated VM called a *driver domain* (shown in Figure 2.1). The driver domain is given direct access to the hardware and performs I/O operations on behalf of the guest VMs. So the driver domain also hosts the physical device drivers needed to access the I/O devices. During network I/O, all packets traverse the driver domain to be either forwarded to the physical device or delivered to a destination domain. Xen [8, 24], the L4 microkernel [25], and Microsoft Hyper-V [9] are examples of hypervisors that use this I/O model.

The primary benefit of this model is that the driver domains provide a safe execution environment for the device drivers, the biggest source of OS bugs [26]. This ensures that any bugs in the device drivers are contained within the driver domain

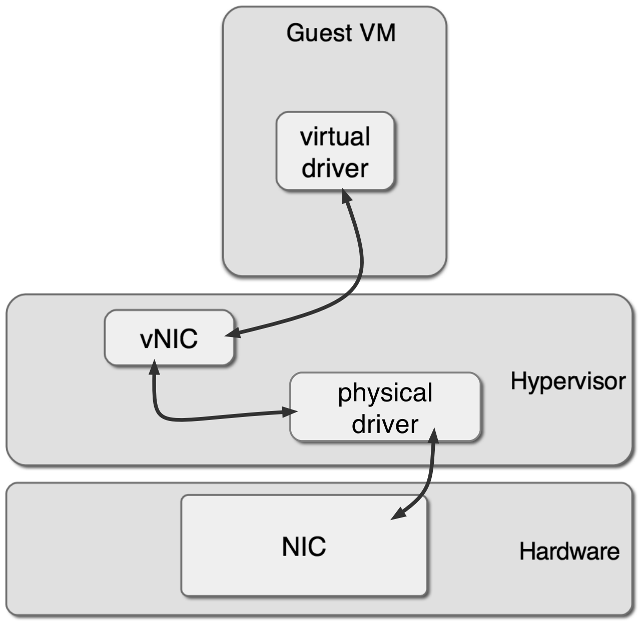


Figure 2.2 : *Hypervisor-based I/O model. The hypervisor implements the virtual devices and also hosts the physical device drivers.*

and cannot corrupt or crash the hypervisor and the other VMs running in the system. Although driver domain crashes can still affect the guest VMs due to interrupted I/O service, this is a more tolerable failure mode. It usually only lasts a short period of time since I/O service can be rapidly restored by simply rebooting a faulty driver domain.

Also, the driver domain runs a largely unmodified operating system. Consequently, it is able to use all of the device drivers that are available for that operating system. This greatly simplifies the complexity of providing support for a wide variety of devices in a virtualized environment. Further, it minimizes the cost to develop and maintain new device drivers.

While the driver domain model provides several desirable properties, it also incurs

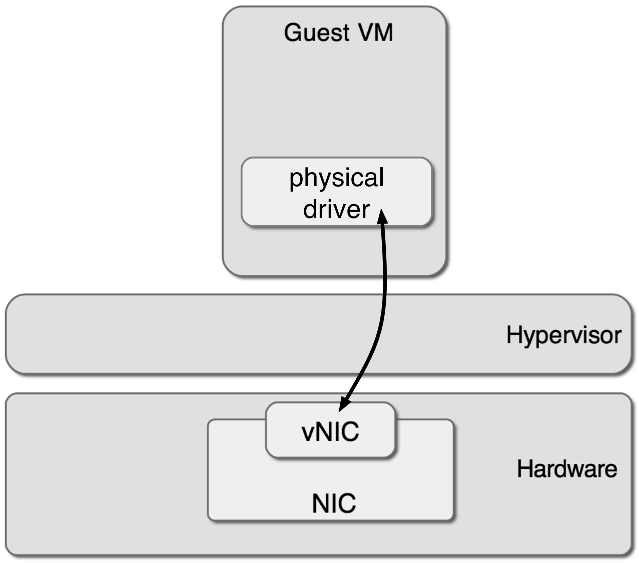


Figure 2.3 : *Direct I/O model. The I/O device is logically partitioned into multiple virtual devices. The guest VM hosts the physical device drivers and has direct access to the device.*

significant CPU overheads. One of the major sources of overhead is the mechanism that is used to share memory between VMs. The memory sharing mechanism is needed to allow the driver domain to access the guest I/O buffers. For example, Xen uses the *grant mechanism* to support the sharing of memory.

2.2.2 Hypervisor-based I/O Model

In this I/O model, the virtual devices are implemented, again in software, but within the hypervisor (shown in Figure 2.2). Further, the hypervisor also hosts the physical device drivers needed to access the I/O devices. VMware's ESX server [10] and Linux KVM [11] are examples of hypervisors that supports this I/O model.

The hypervisor has access to all guest VMs' memory. Therefore, this I/O model does not incur any of the memory sharing overheads. Instead, it sacrifices fault isolation for better performance. Since the device drivers are hosted within the hypervisor, a device driver bug can potentially crash the entire system or corrupt the hypervisor itself. Moreover, this model also increases the size of the trusted computing base (TCB).

2.2.3 Direct I/O Model

In this I/O model, the virtual devices are implemented in hardware by the I/O devices [12–16] (shown in Figure 2.3). In other words, the I/O devices are logically partitioned into multiple contexts which present virtual device interfaces to individual VMs. So the guest VMs can directly communicate with the I/O device bypassing any software intermediary. Hence these devices are also called *pass-through* devices. Today, there exists an industry-wide standard called *single root I/O virtualization (SR-IOV)*, which has been adopted by several network interface vendors to implement this I/O model [27–30].

The primary benefit of this I/O model is that it eliminates most of the software overheads, and therefore, supports near-native performance. But, unlike the software I/O virtualization models, this model lacks support for fault isolation and device transparency. In particular, direct I/O requires device-specific code in the guest VM which has several negative consequences. It increases guest image complexity, reduces guest VM portability, and complicates live guest migration [31–33] between systems with different devices. Moreover, devices which support this I/O model only provide a limited number of virtual contexts. Therefore, scalability can also be an issue.

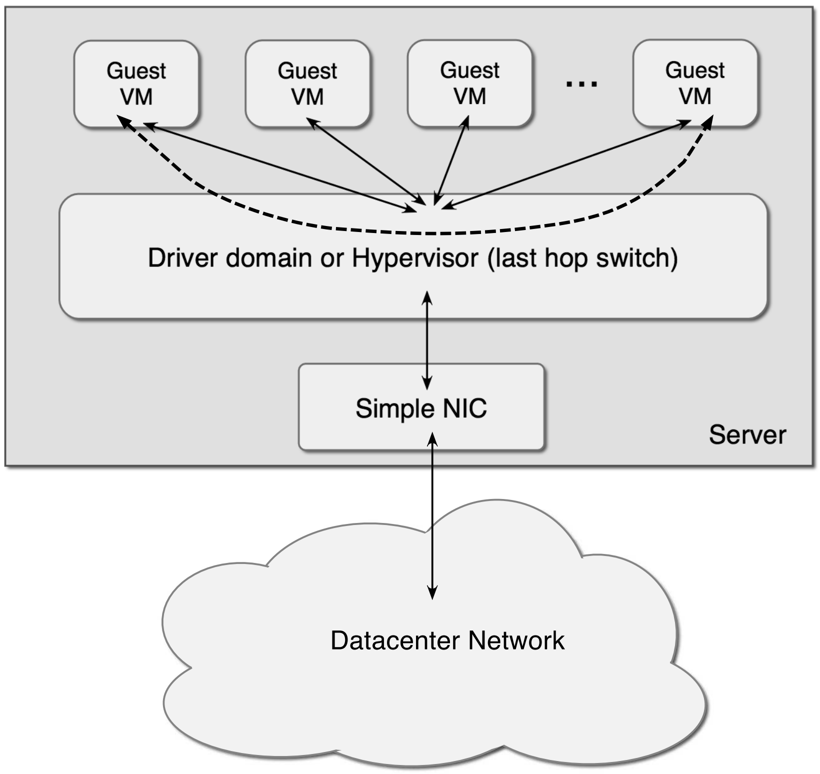


Figure 2.4 : *Purely software approaches for last hop virtual switching. These architectures rely on software–either the hypervisor or a driver domain—to virtualize a simple standard NIC. As the dashed arrow shows, the packet switching happens entirely in the software intermediary.*

2.3 Last Hop Virtual Switching Architectures

The current state-of-the-art last hop virtual switching solutions for datacenter servers can be classified into three main categories.

2.3.1 Purely Software Approaches

The first category of systems (shown in Figure 2.4) implement the last hop virtual switch completely in software within the server. Typically, this solution is used in systems which include a simple NIC that is virtualized by a software intermediary, either using the hypervisor-based or the driver domain I/O model. The Linux bridge [18] and VMware's vSwitch [17] are examples of software switches used in such systems. Cisco and VMware have also made this last hop switch look and behave similarly to other switches in the datacenter [19]. Further, since these implementations are in software, they tend to have a rich set of packet processing functionalities such as ACL matching and link-scheduling.

However, the purely software approaches for last hop virtual switching cannot sustain high throughput for three reasons. First, the cost of supporting advanced switching functionalities like packet filtering using conventional approaches can be expensive in software. Second, regardless of how expensive the packet processing itself is, merely getting the packet to and from the software intermediary can be very resource intensive [34]. Third, parallelizing these software implementations to take advantage of multiple processor cores remains challenging; it has been shown that even a judicious mapping of multiple driver domain threads to cores can often result in a net throughput *loss* [35].

Recently, flow-based switching has been used to address the first of these three issues. The fundamental idea is that the packets are switched on a per-flow basis instead of the conventional per-packet switching. This can have a substantial impact on performance since many operations, such as packet filtering, can then be performed per-flow. As a result, the software overheads due to these operations can be significantly reduced. Open vSwitch [36] is an example of a software switch which

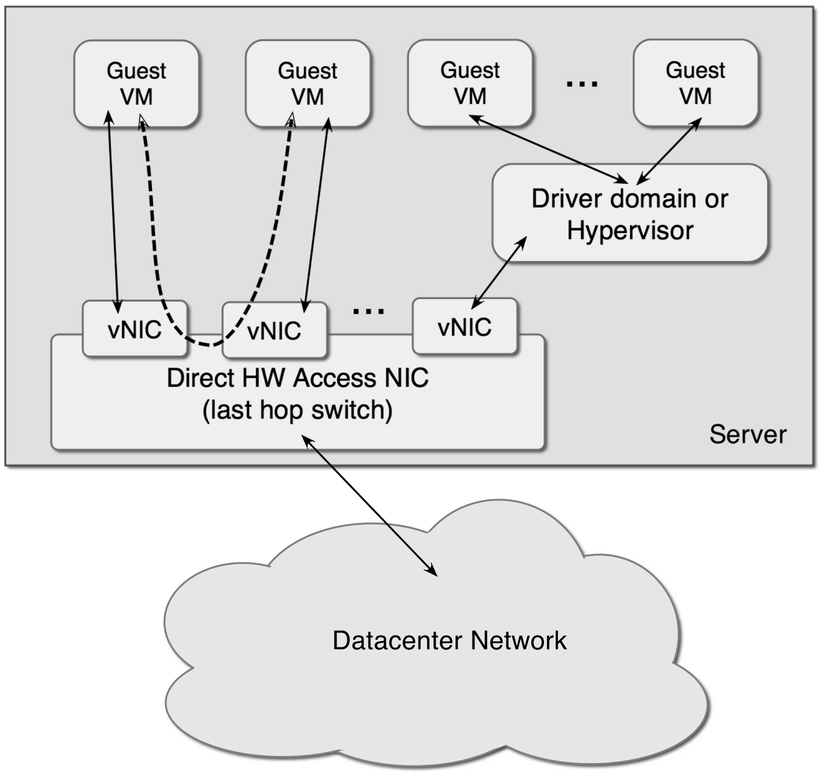


Figure 2.5 : *Network interface-based approaches for last hop virtual switching. These architectures employ sophisticated NICs that allow a subset of the VMs to directly access the hardware and support rudimentary switching. As the dashed arrow shows, the packet switching happens entirely inside the NIC.*

implements flow-based packet switching.

2.3.2 Network Interface-based Approaches

The second category of systems (shown in Figure 2.5) employ more sophisticated NICs which implement the direct I/O model. These NICs also implement switching internally, that is within the hardware. However, today most of them only implement a rudimentary form of a switch that does not support any advanced switching features.

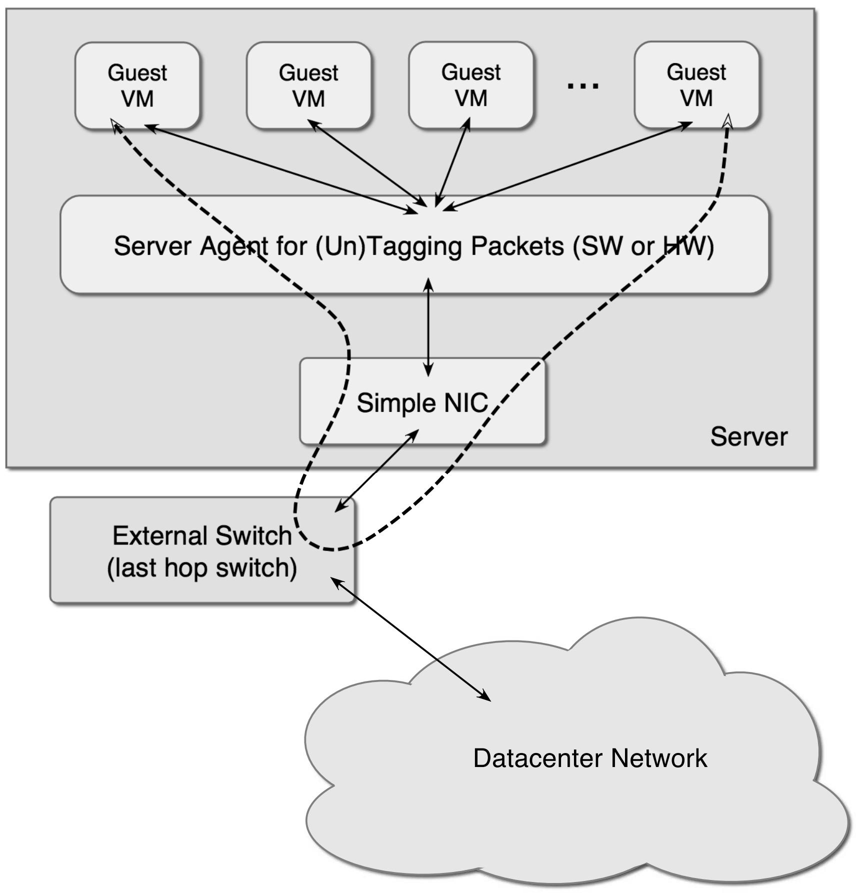

Guest VM
Guest VM
Guest VM
...
Guest VM
Server Agent for (Un)Tagging Packets (SW or HW)
Simple NIC
Server
External Switch (last hop switch)
Datacenter Network

Figure 2.6 : *External switching approaches for last hop virtual switching. The servers blindly forward* all *packets to the external switch which then manages the traffic on a per-VM basis to ensure isolation and QoS guarantees.*

While features like packet filtering using TCAMs are being added to some of these NICs [29], such solutions will neither be scalable nor cost-effective. Further, these NICs waste substantial I/O bandwidth while switching inter-VM packets because they always transfer the full packet payload twice over the I/O bus (to and from the NIC).

Another network interface-based approach for last hop switching involves multi-queue NICs, such as Intel's 10 GbE VMDq NICs [34, 37]. Multi-queue NICs can be used to accelerate the purely software-based network I/O virtualization models. Unlike a direct I/O NIC, a multi-queue NIC does not allow direct access by guest VMs. Instead, it can be used by a driver domain or the hypervisor to allocate a unique hardware TX/RX queue to each guest VM. Then the NIC de-multiplexes the incoming packets and directly DMAs them to the destination VM. Essentially, the NIC implements a very simple switch, which suffers from the same disadvantages as with the direct I/O NICs.

2.3.3 External Switching Approaches

The third approach tries to leverage the functionalities that already exist in today's datacenter switches. This approach uses an external switch for switching *all* packets including those belonging to inter-VM network traffic (as shown in Figure 2.6). In this architecture, a server agent and the external switch attach a special label to each packet that identifies the VM the packet belongs to. While the server agent uses this label to de-multiplex the packets into per-VM receive queues, the external switch uses it to enforce per-VM access controls and QoS. This also simplifies management, since *all* traffic from within the server now transits a traditional switch and hence can be managed by a network manager system. Today, there are two competing standards to implement this approach—Virtual Ethernet Port Aggregator (VEPA) [20] and VN Tagging [21].

Fundamentally, this approach results in a wastage of network bandwidth since even packets from inter-VM traffic always travel all the way to the external switch. Further, similar to the network interface-based approaches, this approach can also result in a wastage of I/O bus bandwidth.

Today, there are not many systems of this kind available for experimentation, however, a server agent implemented in software is very likely to incur a good fraction of the CPU overhead of the software-based approaches discussed above (category 1). In particular, the packets have to still traverse either the hypervisor or a driver domain. However, a server agent implemented in hardware, *i.e.,* within direct I/O NICs, can potentially eliminate these overheads.

Chapter 3

Rethinking Memory Sharing with I/O Devices

3.1 Introduction and Motivation

In I/O virtualization architectures that use driver domains, memory sharing during I/O operations occurs in two levels, as shown in Figure 3.1. First, I/O buffers in guest domains' memory have to be safely shared with virtual I/O devices implemented in the driver domain. This allows the driver domain to perform I/O operations on behalf of the guest virtual machines (VMs). For example, during network I/O, the driver domain needs write access to the I/O buffers in a guest VM's memory so that it can copy the contents of packets arriving for that guest VM. Similarly, for packets that are transmitted by a guest VM, the driver domain needs read access to the guest I/O buffers so that it can parse the packet headers and determine where to route them. Second, the I/O buffers have to be safely shared with physical I/O devices. For example, during network I/O, the I/O device needs DMA write access to the I/O buffers so that it can copy the contents of incoming packets into them. Similarly, for packets that are transmitted, the I/O device needs DMA read access to the I/O buffers so that it can copy the contents of the packets from them.

In the Xen virtualization platform, the first level of memory sharing between VMs is supported using the *grant mechanism* [8, 24]. The grant mechanism allows a source domain to control which of its memory pages can be accessed by a specified destination domain. In addition, it allows the destination domain to validate that the shared memory pages belong to the source domain. During I/O, the grant mechanism is used by the guest domain to *grant* the driver domain access to its I/O buffers.

The second level of memory sharing with I/O devices is supported using *I/O Memory Management Units (IOMMUs)* [38,39]. The IOMMUs are used to perform address translation and validation of all memory accesses from devices, through DMA, using IOMMU tables (I/O page tables). A DMA operation fails if a valid translation (mapping) does not exist in the IOMMU table or if a valid translation exists but the access permissions are not sufficient. Thus the IOMMUs can protect against incorrect or malicious memory accesses by the I/O devices. In Xen, the IOMMU mappings are setup and torn down during the grant operations from the driver domain [40].

Previous work [34,41] has shown that the grant mechanism incurs significant overhead when performing network I/O, and has also shown that most of this overhead is incurred in the driver domain. This is mostly due to the overheads of grant hypercalls and of the high cost of page mapping/unmapping operations executed in these hypercalls. For instance, in the experiments, the memory sharing overheads accounted for nearly 60% of the CPU cycles consumed in the driver domain while processing network packets and the driver domain CPU was a performance bottleneck in all the experiments. Consequently, this limited the rate at which a guest domain can transmit/receive packets since it was not able to utilize the processor to the maximum extent possible. Additionally, the setting up and tearing down of IOMMU mappings further increased this overhead.

A *grant reuse scheme* can greatly reduce the number of grant issue and revoke operations that are needed for I/O by taking advantage of temporal and/or spatial locality in a guest domain's utilization of I/O buffers. The guest domain can issue a grant for a page containing I/O buffers, then use the page several times for I/O, and

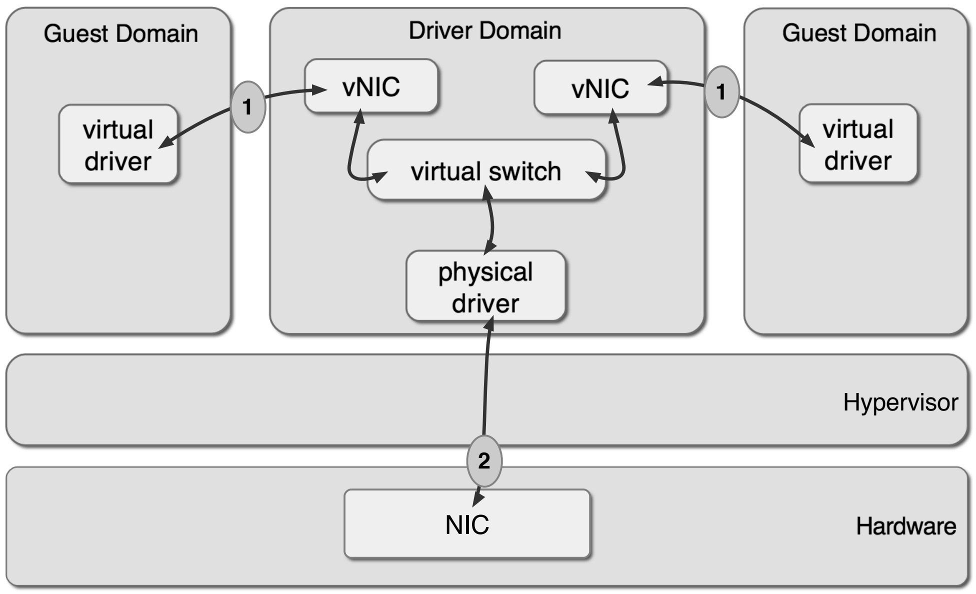


Figure 3.1 : *Two level memory sharing in driver domain I/O model. First, I/O buffers are shared with the virtual device (vNIC). Second, they are shared with the physical device (NIC).*

finally revoke access to that page. In contrast, in the existing implementation every I/O involves grant issue and revoke operations. So the grant reuse scheme reduces the number of grant hypercalls and page mapping/unmapping operations needed for I/O. Further, the IOMMU mappings can also be reused when the corresponding grants are reused.

To support the grant reuse scheme, a new mechanism is proposed that replaces the existing grant mechanism in Xen. Whereas the existing grant mechanism requires the guest domains to coordinate with the driver domain to revoke a grant, the key idea of the new mechanism is to enable the guest domains to *unilaterally* issue and revoke grants. By breaking this dependency, the new mechanism avoids the need for a handshake protocol between the guest and driver domains to revoke the grant to a page, as would be needed with the existing grant mechanism. More generally, using the new mechanism to control memory sharing between two arbitrary guest domains has the advantage that each guest domain can stop sharing its pages with its peer at any time. In particular, each guest domain can forcibly revoke its grants in case its peer misbehaves.

Additionally, the new mechanism provides a *unified interface* that can extend the control of memory sharing to I/O devices using the IOMMU hardware, for both pass-through device access (*i.e.,* direct I/O) [13,27] and when using an intermediary driver domain [8].

While this work only explores the use of the new mechanism for network I/O, we believe that it can completely replace the existing mechanism in Xen. The new mechanism is no less general than the existing mechanism in Xen, and the ability to unilaterally revoke grants provides greater robustness against non-cooperative peers. In general, the new mechanism is applicable to any driver domain-based I/O architecture.

The rest of this chapter is organized as follows. Section 3.2 describes how memory is shared between domains using the existing grant mechanism in Xen. Section 3.3 describes how memory is shared with pass-through devices. Section 3.4 describes the new mechanism and Section 3.5 presents the grant reuse scheme under the new mechanism. Finally, Section 3.6 presents an evaluation of the grant reuse scheme when performing network I/O operations.

3.2 Memory Sharing in Xen

The grant mechanism in Xen allows the driver domain to access the guest I/O buffers in a safe manner and the guest domain to limit the memory pages shared with

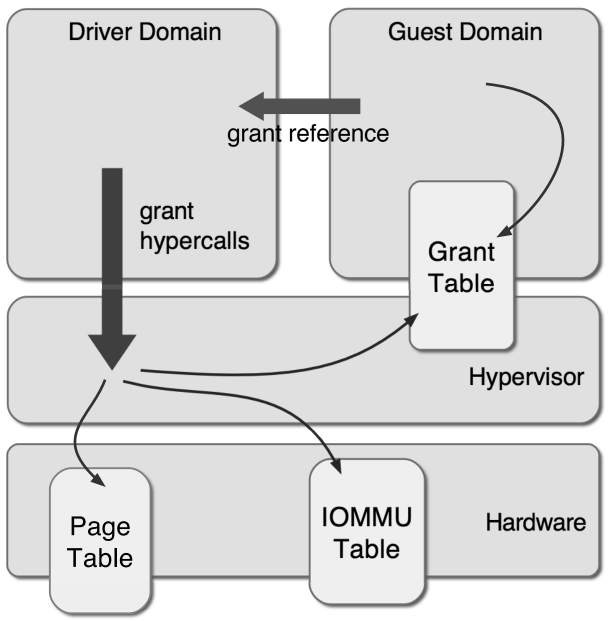


Figure 3.2 : *Memory sharing interface using grant mechanism in Xen. Guest domains interact indirectly with the hypervisor through the grant table. The driver domain interacts directly with the hypervisor through grant hypercalls.*

the driver domain to only those containing the buffers being used for I/O operations.

Creating Shared Memory: The existing memory sharing interface using the grant mechanism is illustrated in Figure 3.2. A guest domain shares one of its memory pages in two stages. In the first stage, the guest domain simply indicates the page it desires to share as follows:

- First, the guest domain allocates a *grant reference* for that memory page. The grant reference points to a unique entry in a *grant table*, which is shared between the guest domain and the hypervisor.

- The grant entry contains the shared memory page address, the driver domain id, and the access permissions (read-only or read-write). The guest domain fills

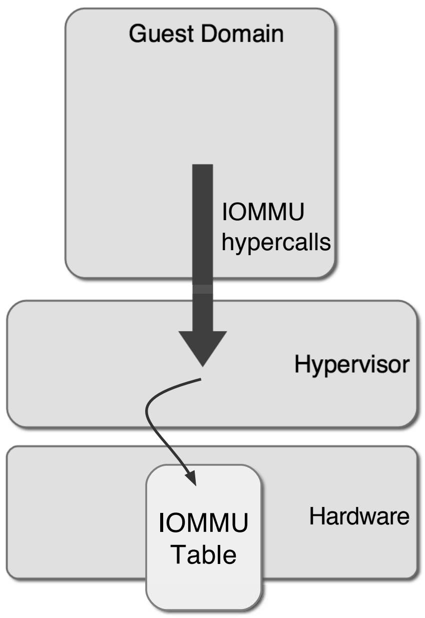


Figure 3.3 : *Memory sharing interface when using pass-through devices. Guest domains interact directly with the hypervisor to manipulate the IOMMU table.*

the grant table entry using simple memory writes.

The guest domain then passes the grant reference to the driver domain via Xen's inter-domain network I/O channel called *net-channel*. In the second stage, the driver domain uses the grant reference to access the shared memory. This stage requires hypervisor intervention. This involves the following steps:

- The driver domain issues a grant hypercall, to enter the hypervisor, passing the grant reference and a virtual address as arguments.

- The hypervisor first checks whether the grant reference is valid. It reads the guest domain's grant table entry and checks whether the domain that invoked the hypercall is the intended destination. It then obtains the machine address of the shared page and checks if that page is owned by the guest domain.

- If all the tests pass, the hypervisor then pins the memory page and maps the page within the driver domain's address space at the given virtual address. This requires adding a new entry (driver domain virtual address $\rightarrow$ guest domain machine address) to the driver domain's page table.

- Finally, the hypervisor adds an entry to the IOMMU table. This entry is an identity mapping (guest domain machine address $\rightarrow$ guest domain machine address).

- Now the driver domain and the I/O device can safely access the shared memory page.

Page pinning ensures that the page ownership does not change while a page is shared. Otherwise, memory corruption is possible. Consider the scenario where a guest domain gives up a shared page to the hypervisor, without the driver domain's knowledge. The hypervisor might then allocate this page to another guest domain. Now if the driver domain inadvertently copies an incoming packet into that page, it will end up corrupting the other guest domain's memory.

Revoking Shared Memory: A shared memory page is revoked, again, in two stages. In the first stage, the driver domain stops accessing the shared page as follows:

- The driver domain issues another grant hypercall, to enter the hypervisor, passing the grant reference as an argument.

- The hypervisor first removes the IOMMU mapping. It also performs the required IOTLB invalidation.

- Then it unmaps the page from the driver domain's address space and unpins the shared memory page. It also performs the required TLB invalidation.

Subsequently, in the second stage, the guest domain revokes the shared page as follows:

- It invalidates the corresponding grant table entry (again, through simple memory writes).

In the standard network I/O model in Xen, the guest domain creates grants for its receive/transmit I/O buffers to provide shared access to the driver domain. During packet transmission, the driver domain first issues the grant *map* hypercall. Once the I/O has completed, the driver domain issues the grant *unmap* hypercall. Then the driver domain notifies the guest domain that the I/O operations have completed. This also serves as a notification that the guest domain can revoke the grant for the corresponding page.

During packet reception, the incoming packets are copied into local driver domain buffers first. Once the destination of the packet is determined, the packets are copied into that guest domain's I/O buffers. So, unlike packet transmission, the guest I/O buffers are not shared all the way to the I/O device. Further, typically the I/O device is given access to all of driver domain's memory during initialization. So IOMMU mappings are not setup during packet reception. Once a packet is received, the driver domain performs a single grant *copy* hypercall. The hypervisor then validates the grant and pins the page. Then it copies the packet and finally, unpins the page. Then as before, the driver domain notifies the guest domain and the guest domain revokes the grant.

Essentially, a grant is issued and revoked for each and every I/O operation, leading to significant performance overhead for memory sharing. Some of these overheads, especially the cost of issuing hypercalls, can be reduced by batching the grant operations. But despite this optimization, the overheads remain high.

3.3 Memory Sharing with Pass-through Devices

The main feature of pass-through devices is that they bypass the driver domain and have direct access to the hardware. Here, a guest domain directly shares its I/O buffers with the I/O device. In this scenario, an IOMMU is essential to restrict the device's access only to the memory of the guest domain to which it is assigned. Otherwise, memory isolation cannot be enforced since a malicious guest domain can setup DMA operations to other guest domains' memory.

In Xen, each pass-through device is configured with an IOMMU table which is used to provide *coarse-grained protection*. In this mode, the IOMMU table is configured with valid translations for exactly all the pages that belong to the guest domain accessing that device. So the IOMMU table is mostly static and does not change unless the set of pages assigned to the guest domain changes.

To provide a higher level of protection against buggy device drivers or to enable user-level drivers, the set of valid IOMMU translations can be limited to only a small set of pages which contain I/O buffers that need to be accessed by the device. In this mode, which is referred as *fine-grained protection*, the guest domain needs to invoke the hypervisor so that it can add the corresponding page mappings to the IOMMU table before programming a device DMA operation, and remove the page mappings after the DMA operation is completed. (Figure 3.3) and this can incur significant overheads [40, 42].

3.4 The Design

A grant reuse scheme can significantly reduce grant overheads by reusing the same grant for multiple I/O operations. A guest domain can issue a grant for a page containing I/O buffers, then use the page several times for I/O, and finally revoke access to the page. Thus under the reuse scheme the overheads of the grant hypercalls and the mapping/unmapping operations are not incurred on every I/O operation.

In the grant reuse scheme, the domain initiating the sharing (the source domain) should be able to revoke a grant at any given time. For example, suppose a guest OS shares a page with a driver domain for network I/O, and then later the guest OS re-purposes the page, say to assign the page to a user-level process. Before re-purposing the page, the guest OS might want to revoke the grant to prevent subsequent access to the page by the driver domain. In general, a source domain should have the flexibility to revoke a grant from various OS subsystems running in that domain. Using the existing grant mechanism in Xen, this would require the source domain to carry out a protocol handshake with the destination domain via an inter-domain I/O channel (like net-channel) to revoke the grant. This handshake protocol prevents the source domain from completing the grant revocation until the destination domain unmaps the page and notifies the source domain that the unmapping is complete. If the destination domain, for some reason, is unable to respond, there is no way for the source domain to revoke access to the shared page.

Instead, a new mechanism is proposed that breaks this dependency by allowing the source domain to unilaterally issue and revoke grants. This means that the guest domain can issue and revoke grants, during I/O, using a simple hypercall interface without requiring driver domain participation. This avoids the handshake protocol completely. Moreover, the new mechanism has the benefit of reducing the trust required between any two domains that are sharing memory. Either guest domain can unilaterally remove access privileges to its pages from the other guest domain, without requiring the cooperation of the other guest domain to unmap the pages.

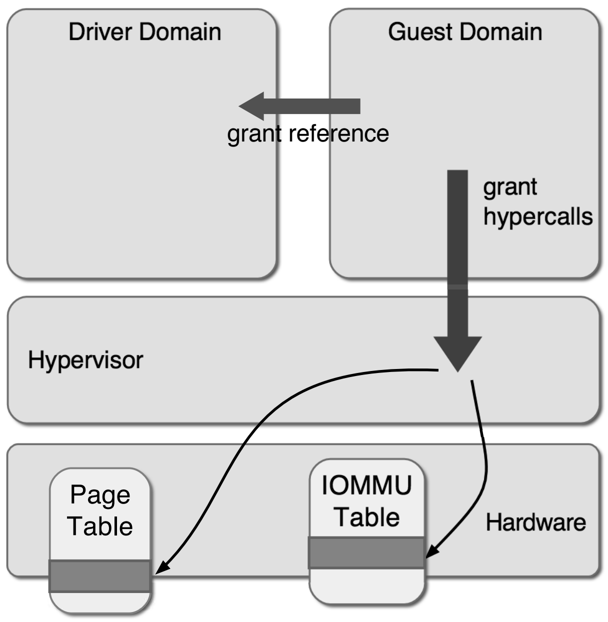


Figure 3.4 : *New unified memory sharing interface. Guest domains interact directly with the hypervisor, via grant hypercalls, to issue and revoke grants.*

This is particularly useful in case the other guest domain misbehaves.

It turns out that the guest interface to the new mechanism is very similar to an interface needed to add and remove page mappings from an IOMMU table when using the fine-grained protection mode. Thus the grant interface can be used to issue/revoke grants and/or add/remove entries to/from an IOMMU table. This unification simplifies I/O support in the guest OS, to share memory either with driver domains or with devices directly. For example, for guest domains running Linux this can be supported by a common implementation of the DMA API interface [40, 42]. Additionally, this allows the same "grant" reuse scheme to be used both for driver domains and directly accessed I/O devices, providing performance benefits in both cases.

The new memory sharing interface is illustrated in Figure 3.4. In the proposed

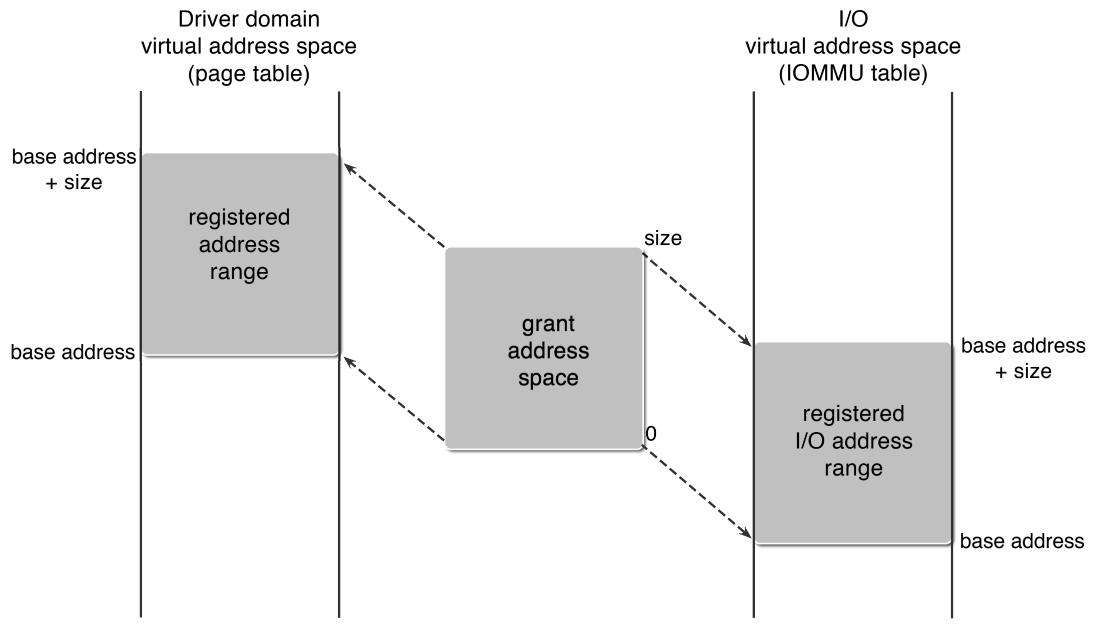


Figure 3.5 : *Grant address space. The guest pages are mapped within the registered virtual and I/O virtual address ranges.*

mechanism, the guest domain directly interacts with the hypervisor, via hypercalls, to issue and revoke grants. Further, a common interface is presented for both pass-through and standard I/O devices. The subsequent sections explain the design in greater detail.

3.4.1 Initialization

The initialization occurs in one or two stages depending on whether the guest domain has direct access to the I/O device or not. In the first stage, the destination domain (driver domain) issues a hypercall to begin initialization. The driver domain passes a virtual address range in its address space (base virtual address and size) and a device id as arguments to the hypervisor. The hypervisor registers this virtual address range for the specified device. The driver domain can also, optionally, specify a different I/O virtual address range in the IOMMU's address space. But the size of both the address ranges must be identical. This stage is not required if the guest domain has direct access.

In the second stage, the guest domain issues a hypercall passing the device id as an argument. There are two cases here:

- If the device id corresponds to a physical device and if that device has been directly assigned to that domain, then the hypervisor initializes an IOMMU table for that domain/device pair. The hypervisor specifies a *grant address space*, which, in this case, is same as the IOMMU's address space.
- If the device id corresponds to a virtual device, then the hypervisor checks if the corresponding driver domain has performed the registration. If so, it uses the size of the registered address ranges to specify the grant address space (*i.e.,* $0 \rightarrow$ size of the address range).

The grant address space is illustrated in Figure 3.5. Finally, the hypervisor returns a handle back to the guest domain which is used by it for all future grant operations.

3.4.2 Creating Shared Memory

Shared memory is created using the new interface as follows:

- The guest domain issues a grant hypercall, to enter the hypervisor, passing the handle, the address of the page to be shared, the access permissions, and a grant reference. The notion of a grant reference is retained from the existing mechanism. But under the new mechanism, the grant reference is an offset within the grant address space.

- The hypervisor first obtains the machine address of the shared page and validates that the page is owned by that guest domain. If the validation succeeds, it pins the page.

- Then the hypervisor maps the page within the registered virtual address range in the driver domain's address space. The hypervisor computes this virtual address as the sum of: (a) the base address of the registered virtual address range, and (b) an offset equal to the grant reference. This requires adding a new entry (computed virtual address $\rightarrow$ guest domain machine address) to the driver domain's page table. This step is not needed when using pass-through devices.

- Finally, the hypervisor adds a mapping within the registered I/O virtual address range. It again computes the I/O virtual address as the sum of: (a) the base address of the registered I/O virtual address range, and (b) an offset equal to the grant reference. This again requires adding a new entry (computed I/O virtual address $\rightarrow$ guest domain machine address) to the IOMMU table.

The guest domain passes the grant reference to the driver domain via the net-channel. The overhead of using grants in the driver domain is very low since the guest pages are already mapped. Now the driver domain can obtain the virtual address to access the guest page by using the grant reference as an offset within the virtual address range it registered with the hypervisor. Similarly, it can also obtain the I/O virtual address to setup DMA operations by using the grant reference as an offset within the I/O virtual address range. Essentially, the driver domain has to only perform simple arithmetic operations.

3.4.3 Revoking Shared Memory

Shared memory is revoked using the new interface as follows:

- The guest domain issues another grant hypercall, to enter the hypervisor, passing the handle, and the grant reference corresponding to the shared page to be revoked.
- The hypervisor first removes the IOMMU mapping. It also performs the required IOTLB invalidation.
- Then it unmaps the page from the driver domain's address space. It also performs the required TLB invalidation. This step is not needed when using pass-through devices.
- Finally, it unpins the shared memory page.

So a guest domain can use the new memory sharing interface to revoke a grant, *at any time*, without any cooperation with the driver domain. While this would work properly under normal error-free conditions, it can lead to problems when a guest domain misuses the new interface. In particular, consider the scenario when a buggy or malicious guest domain revokes a grant while the corresponding page is still being used for an active I/O operation. Even under such conditions the grant revocation will succeed, and as a result, the guest page will not be accessible to neither the driver domain nor the device. But since the driver domain is not aware of the revocation, it might inadvertently try to read or write the page. It might have also programmed the device to DMA packets to or from that memory page. But since the guest page is no longer mapped in the driver domain's address space or the IOMMU's address space, these memory accesses will result in a fault—a page fault or an IOMMU fault.

Such faults might render the driver domain unusable. Though the driver domain can be restarted under such conditions, the resulting I/O interruption it far from ideal.

A potential solution is to use *dummy* pages to replace the real pages when their grants are revoked by a guest domain. So the hypervisor allocates a dummy page for every device—virtual or physical—in the system. Now when the grant to a page is revoked by a guest domain, the associated dummy page is atomically swapped into both the driver domain's and the IOMMU's address pace. Therefore, any future accesses to that page will no longer result in a fault. But the driver domain and the device might have to deal with corrupted packets, and typically, they are robust enough to do so.

3.5 Reuse of Grants

The cost of using the grant mechanism can be reduced by enabling the reuse of the same grant across multiple I/O operations, taking advantage of locality in the use of I/O buffers, whether spatial locality—multiple I/O buffers sharing a page—or temporal locality. This allows a grant acquired for a guest domain's memory page to be reused for future I/O operations. A grant associated with a guest domain's memory page can be revoked at any given time.

Grant reuse is effective at reducing grant overhead only if the reused page remains mapped in the driver domain's address space since page mapping and unmapping are expensive operations. When the guest domain desires to revoke a grant, the page must be unmapped from the driver domain's and IOMMU's address spaces. In the proposed mechanism, this can be done unilaterally by the guest domain without driver domain's cooperation. In contrast, in the original grant mechanism, grants can be revoked only with the cooperation of the driver domain which has to issue the hypercall to unmap the guest pages from its address space.

While the new mechanism enables simple and unilateral grant revocation under the grant reuse scheme, the performance benefits are predominantly provided by the grant reuse scheme itself.

3.5.1 Reuse Scheme

A guest domain can employ different schemes to reuse grants for minimizing the grant related overheads. These schemes are essentially a trade-off between performance and security. Coarse-grained protection, where all the guest pages are shared persistently, provides the best performance but the least security. This scheme is currently used with pass-through devices in Xen. Strict fine-grained protection, where a guest page is only shared for the duration of a single I/O operation, provides the best security but the least performance. This scheme is used under the standard network I/O model in Xen. Several other schemes have been proposed in the past, such as shared mappings, delayed invalidations, and optimistic tear down [43, 44]. All these schemes use a relaxed fine-grained protection mode to achieve better performance, without necessarily resorting to the coarse-grained protection mode.

In this implementation, the default reuse scheme was to just limit the number of pages shared by a guest domain at any given time. The size of the grant address space, as registered by the driver domain, defines the maximum number of pages which can be shared by a guest domain. A guest domain can either choose to use the entire grant address space or use only a part of the address space to further limit the number of shared pages. When the grant address space is full, existing grants are revoked to make space for new grants.

An alternate scheme is to revoke a grant to a page when the page is re-purposed to be used for a non-I/O operation. This scheme is used only for grants associated with I/O buffers shared as read-write with the driver domain (receive I/O buffers). The rationale behind this scheme is that a read-write buffer which is re-purposed can be corrupted if its still shared with the driver domain. This scheme is implemented using the Linux slab cache mechanism, which is used to create a dedicated read-write I/O buffer pool. The I/O buffers are allocated from the pool when needed for receive operations and are returned to the pool afterward. Further, the guest OS revokes the grant associated with a guest domain's memory page when it is released from the slab cache. As an added benefit, recycling buffers from the pool can also promote locality leading to a higher degree of grant reuse.

3.5.2 Tracking Grant Use

While different mechanisms can be used in different guest domains to keep track of grant use, the guest OS in the prototype implementation uses a hash table for each virtual device. Each entry in the hash table corresponds to a grant reference. The table is divided into two halves, one half is used to track read-only grants and the other half is used to track read-write grants. If a grant is issued, the hash table entry contains the guest page frame number (pfn), and a reference counter that records the number concurrent active uses of a grant. The pfn is used to resolve hash collisions and the counter is used to check if a grant is active or not. Each entry is treated as a single 8 byte word to facilitate efficient atomic memory accesses.

The guest domain checks whether a grant already exists for a page by looking up the hash table as follows:

- First, the pfn of the page is fed into a hash function. The hash function returns a *hash set* which is a set of contiguous locations in the hash table which can potentially contain a grant for that page.

- Then each entry in this set is checked to see if it contains a grant for the page. If a grant already exists, then the grant can be reused. So the guest domain increments the reference counter and simply reuses the grant.
- If a grant does not exist, a new grant is created for this page. The first free location in the hash set is selected to track this grant. If none of the locations in the hash set are free, then the first inactive grant in the hash set is revoked and this location is used to track this grant. If even this fails then the operation is aborted in the current implementation. A more sophisticated hash table implementation could avoid this using techniques like rehashing.
- Finally, the guest domain initializes the reference counter and invokes a hypercall to issue the grant to the driver domain.

Therefore, the grant overhead is negligible if a grant already exists.

3.6 Evaluation

This section presents experimental results that quantify the benefits of the newly designed grant mechanism. A prototype of the new mechanism was implemented in the Xen hypervisor and in a para-virtualized (PV) Linux domain. The prototype provides full support for network I/O with driver domains. It also implements both the reuse schemes discussed in Section 3.5.1.

Table 3.1 : *Evaluation - Server Configuration*

H/W Configuration		Processor	2.67 GHz Intel Xeon W3520
		Total processor cores	4 (w/o hyperthreading)
		Total memory	6 GB
		Network Interface	Intel 10 GbE CX4
S/W Configuration	Hypervisor		Xen 4.2[*]
	Domain 0	Operating System	Linux PV dom0 kernel v2.6.18[+]
		Number of vCPUs	1
	Driver Domain	Operating System	Linux PV dom0 kernel v2.6.18[+]
		Number of vCPUs	1
		Memory	1 GB
	Guest VM(s)	Operating System	Linux PV domU kernel v2.6.18[+]
		Number of vCPUs	1
		Memory	512 MB

[*] Obtained from the Xen open source mercurial repository (`xen-unstable.hg changeset 23542:23c068b10923`), as of June 2011.

[+] Obtained from the Xen open source mercurial repository (`linux-2.6.18-xen.hg changeset 1021:7b350604ce95`), as of June 2010.

Table 3.2 : *Evaluation - External Server Configuration*

Hardware Configuration	Processor	2.67 GHz Intel Xeon W3520
	Total processor cores	4 (w/o hyperthreading)
	Total memory	6 GB
Software Configuration	Operating System	Linux kernel v2.6.35

3.6.1 Experimental Setup and Methodology

The netperf TCP stream microbenchmark [45] was used in all experiments to generate network traffic. The experiments included scenarios with network traffic in the transmit (*TX experiment*) and receive (*RX experiment*) directions between a guest domain and a NIC, and network traffic between two guest domains running on the same physical host (*inter-VM experiment*). OProfile [46, 47] was used to determine the number of CPU cycles spent when processing network packets. This involved profiling the driver and guest domain virtual CPUs for `CPU_CLK_UNHALTED` events during the experiments.

In each of the above scenarios, the network throughput and the CPU cost were compared. The CPU cost was measured as CPU cycles consumed per packet (henceforth referred to as *cycles/packet*). This metric was used to evaluate the performance overheads in the experiments. The cycles/packet metric was computed by dividing the total packet processing cost across the total number of estimated full-sized Ethernet packets (1514-bytes) transmitted on the wire in an experiment. Since a streaming benchmark was used, the actual packet sizes varied. But most of the packets transmitted on the wire were found to be standard Ethernet sized. Hence the above methodology was adopted. But since TCP segmentation offload (TSO) was used in the transmit side, the fixed cost associated with processing a large TCP segment has been divided across the constituent full-sized Ethernet packets.

The experiments were run on two server machines connected directly to each other using a 10 Gigabit CX4 Ethernet cable. Both the servers had a 2.67 GHz Intel Xeon W3520 quad-core CPU, 6 GB of memory, and a 10 GbE Intel 82598EB Ethernet NIC. The main server ran Xen. It was configured with up to two PV Linux guest domains, and one dedicated PV Linux driver domain in addition to the privileged management domain (domain 0). The driver domain and the guest domain(s) were each configured with a single virtual CPU. Further, each virtual CPU was pinned to a separate CPU core to eliminate potential issues related to scheduling domains and I/O performance [48]. The driver domain was configured with 1 GB of memory, and each guest domain with 512 MB of memory. The external server ran an Ubuntu distribution of native Linux kernel v2.6.35.7. Also, the CPUs at the external server were never a resource bottleneck in any of the experiments. Tables 3.1 and 3.2 summarize the configuration of the servers.

3.6.2 Experimental Results

Figures 3.6 through 3.11 compare the throughput and the CPU cost between the original grant mechanism and the new grant mechanism in all the three scenarios. In each scenario, the original grant mechanism was divided into four cases as follows:

1. In the first case, the original grant mechanism was used without IOMMUs. Unlike the new mechanism, since the original grant mechanism cannot make any unilateral revocation guarantees, IOMMUs are not fundamentally required. This case represents the best case scenario for the existing grant mechanism.

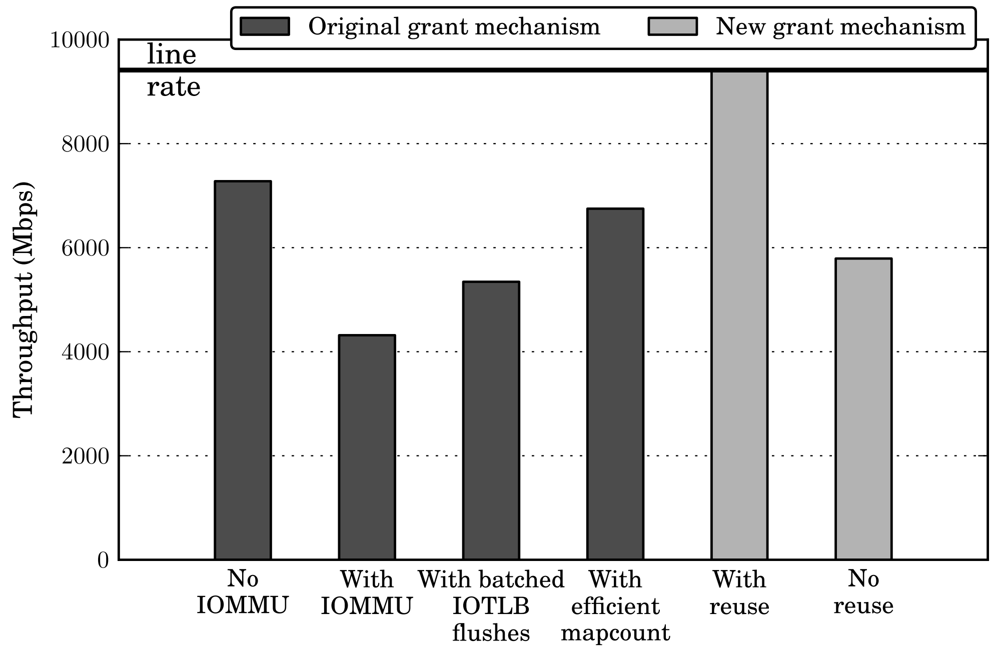


Figure 3.6 : *RX Experiments - Throughput Results*

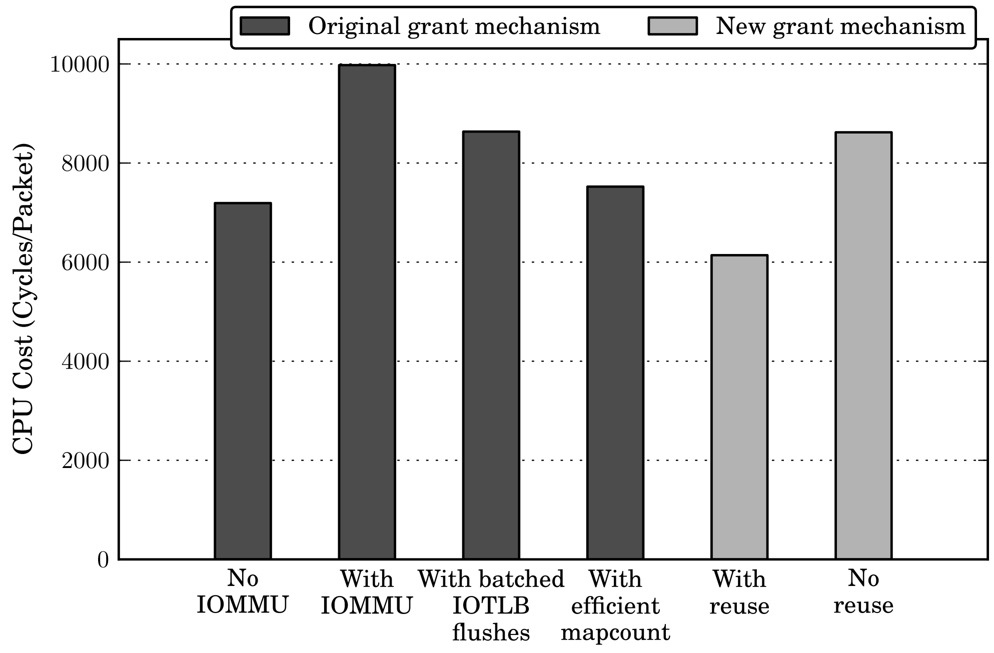


Figure 3.7 : *RX Experiments - CPU Cost Results*

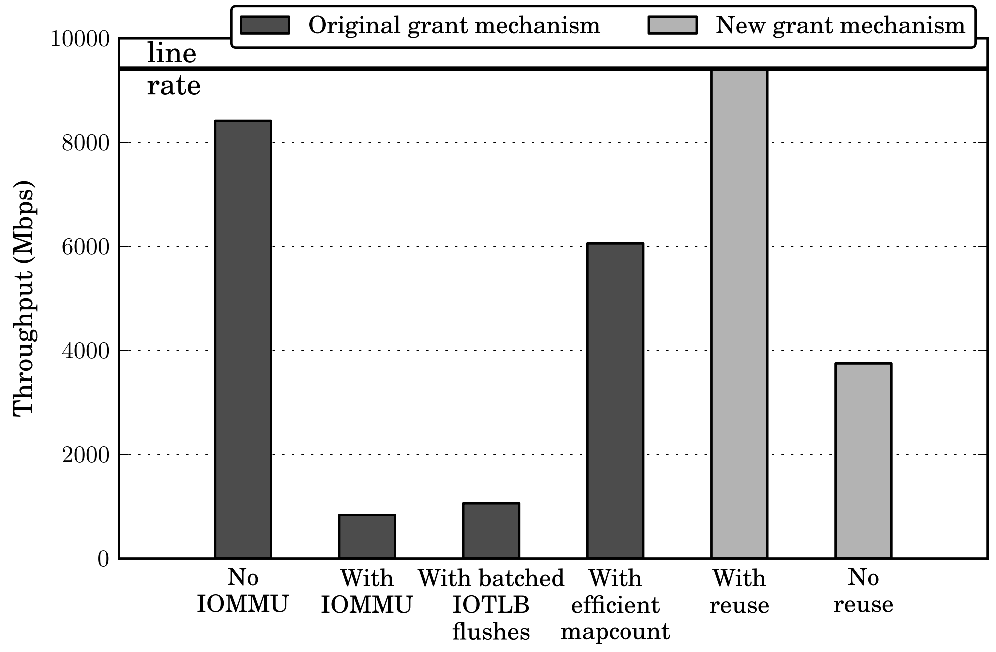


Figure 3.8 : *TX Experiments - Throughput Results*

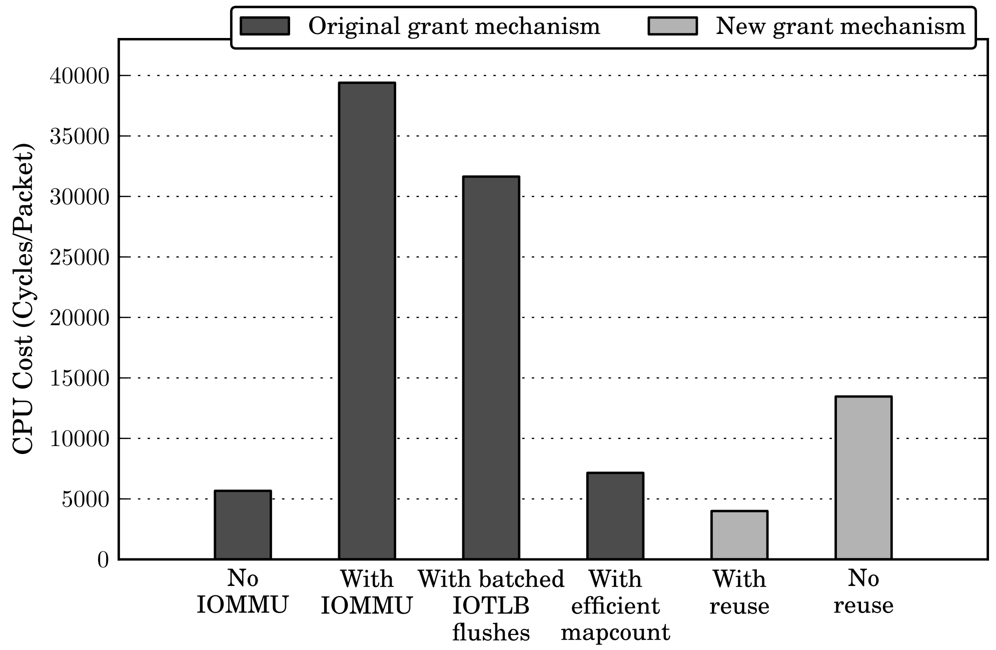


Figure 3.9 : *TX Experiments - CPU Cost Results*

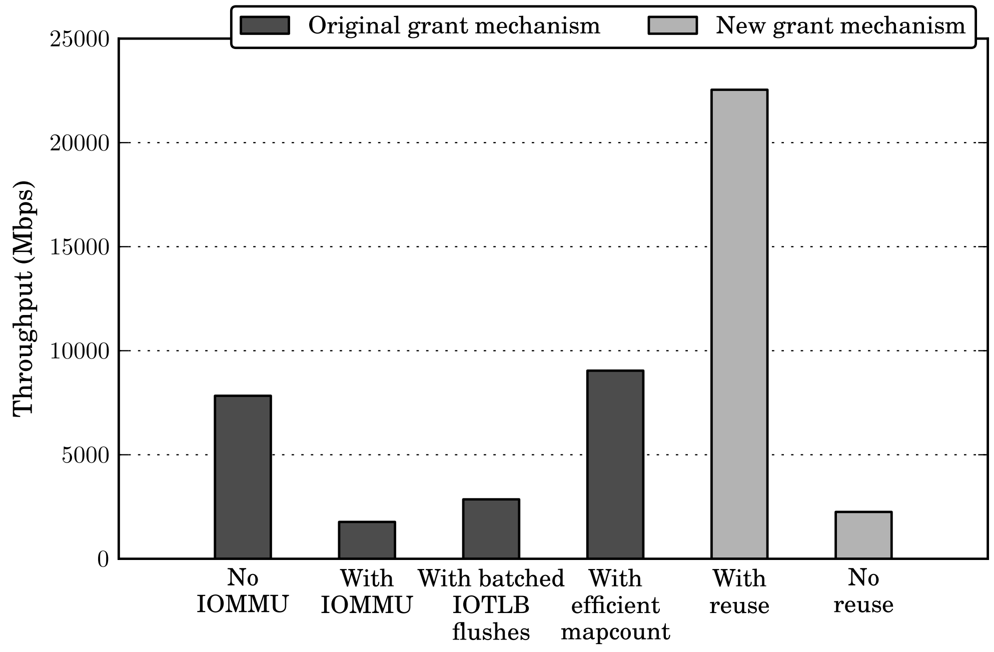


Figure 3.10 : *Inter-VM Experiments - Throughput Results*

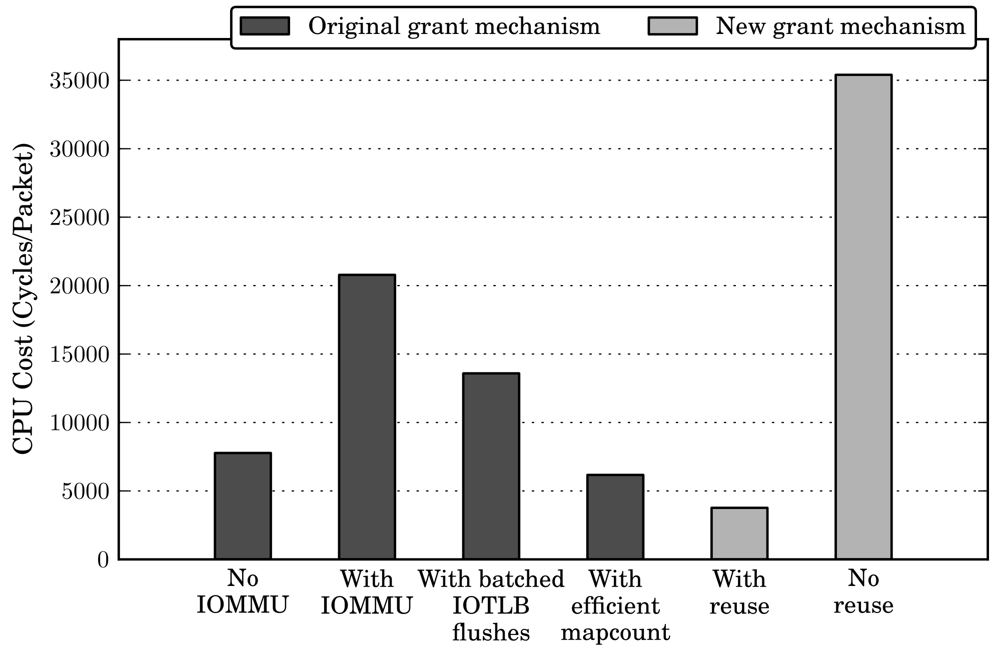


Figure 3.11 : *Inter-VM Experiments - CPU Cost Results*

2. In the second case, the original grant mechanism was used with IOMMUs. This was the base case. Across all the three scenarios, this case resulted in very poor performance with up to 600% increase in CPU cost. This increase was significantly higher than the overheads typically associated with IOMMU operations. This disproportionate increase in the CPU cost (and the corresponding decrease in throughput) is attributed to two primary issues. Since these issues were not fundamental, it was feasible to implement temporary solutions to fix these issues. The next two cases incorporated these fixes. This was done to ensure that the comparison between the new mechanism and the existing grant mechanism was fair.

3. In the third case, *batched IOTLB flushes (invalidations)* was added to the original grant mechanism. The current implementation in Xen performs an IOTLB flush on every unmap call. This is particularly expensive since an IOTLB flush involves polling an IOMMU register to detect completion. Instead, since the grant operations were already batched, this was modified to perform one global IOTLB invalidation after all the operations in a batch were completed.

4. In the fourth case, an *efficient mapcount mechanism* was added to the original grant mechanism (on top of the batched IOTLB flushes). Mapcount is used to compute the number of IOMMU mappings associated with a page. Since the original grant mechanism sets up identity mappings (mfn $\rightarrow$ mfn) in the IOMMU table, an IOMMU mapping is setup for a page only the first time. All further grant map operations for that page do not involve the IOMMU. The mapping is removed only when all the associated grant operations have completed. So mapcount is used to find the number of IOMMU mappings

associated with a page at any given time. Under Xen, every domain has a *maptrack table* that keeps track of all foreign grant mappings within a domain. The current implementation, naively, on *each* grant operation goes through *all* the entries in the maptrack table to compute the mapcount. Depending on the number of entries in the maptrack table this can be a prohibitively expensive computation. Instead, per-page counters were implemented to keep track of the mapcount. This way the mapcount "computation" was avoided on every grant operation.

The TX experiment shows the worst performance when IOMMUs were used under the original grant mechanism. Specifically, it shows a ~600% increase in CPU cost (Figure 3.9) and a ~90% decrease in throughput, from ~8400 to ~800 Mbps (Figure 3.8), as compared with the case where IOMMUs were not used. Unlike the TX experiment, the RX experiment was not as badly affected, with only a ~40% increase in CPU cost (Figure 3.7) and a ~40% decrease in throughput, from ~7200 to ~4300 Mbps (Figure 3.6). There are two reasons for this:

- First, as explained in Section 3.2, IOMMUs are used under Xen only during packet transmission. So in the RX experiment, IOMMUs were used only when the TCP ACKs were transmitted back to the external server. Further, the number of TCP ACKs are typically far fewer as compared with the number of TCP data packets. This means there were far fewer grant map/unmap operations. So the overhead of performing the mapcount computation and the IOTLB flushes were not as high in the RX experiment.
- Second, the number of grant mappings was significantly higher in the TX and inter-VM experiments as compared with the RX experiment. This means that

> the number of entries in the maptrack table was also significantly higher. Since the mapcount computation was proportional to the size of this table, the overhead of computing mapcount was higher in these experiments as compared with the RX experiment.

Hence the RX performance was not as badly affected. In the inter-VM experiment, the driver domain both transmits and receives TCP data packets, so the corresponding performance was between the TX and RX experiments, with a ∼170% increase in CPU cost (Figure 3.11) and a ∼77% decrease in throughput, from ∼7800 to ∼1700 Mbps (Figure 3.10).

The benefit of using batched IOTLB flushes was observed in all the three scenarios. For example, in the inter-VM experiment, the throughput was increased from ∼4300 Mbps to ∼5300 Mbps (Figure 3.10), as a result of a ∼60% decrease in CPU cost (Figure 3.11). The use of efficient mapcount mechanism had an even bigger impact on performance, especially in the TX and inter-VM experiments. For example, in the TX experiment, the CPU cost went down by ∼80% (Figure 3.9) and the throughput went up by ∼470%, from ∼1100 to ∼6000 Mbps (Figure 3.8).

In all the experiments using the original grant mechanism, the driver domain CPU was a resource bottleneck. This limited the rate at which the guest domain could transmit/receive packets since it was not able to utilize its CPU core to the maximum extent possible. A seemingly anomalous result in the inter-VM experiments was that the CPU cost of the original grant mechanism with IOMMUs (optimized) was less than the CPU cost without IOMMUs (Figure 3.11). However, using IOMMUs should have increased the CPU cost due to the following overheads: a) setting up and tearing down the IOMMU mappings and b) performing the IOTLB flushes. But, in this particular case, increased batching of the grant operations was also observed.

In turn, this decreased the CPU cost by reducing the overhead of issuing hypercalls. So the increase in the CPU cost from using IOMMUs was offset by a larger decrease due to batching.

Next, the new mechanism was compared with the optimized version of the original grant mechanism. The new mechanism was divided into two cases:

- In the first case, the grant reuse scheme was used under the new mechanism. This was the base case. So the number of grant issue and revoke operations that were needed for I/O were reduced by taking advantage of temporal and/or spatial locality in a guest domain's utilization of I/O buffers.

- In the second case, the grant reuse scheme was not used. Essentially, grants were issued and revoked on every I/O operation. This was similar to how grants are used under the original grant mechanism. This also represents the worst case scenario under the grant reuse scheme. But typically there is always going to be at least some temporal/spatial locality in a guest domain's utilization of I/O buffers.

In both the RX and TX experiments, the new mechanism with the reuse scheme enabled, achieved line rate ($\sim$9400 Mbps) (Figures 3.6 and 3.8). This was due to a $\sim$20% decrease in the CPU cost in the RX experiment (Figure 3.7) and a $\sim$45% decrease in the CPU cost in the TX experiment (Figure 3.9). In the inter-VM experiment, the throughput increased by $\sim$150%, from $\sim$9000 to $\sim$22500 Mbps (Figure 3.10), due to a $\sim$40% reduction in the CPU cost (Figure 3.11). Unlike the RX and TX scenarios, here the driver domain suffered from grant overheads on both the transmit and receive sides. Since the grant reuse scheme reduced the overhead during both packet transmission and reception, the processing cost and throughput improved more significantly in this case than in the other cases.

When the grant reuse scheme was disabled, the performance under the new mechanism was worse than the performance under the original grant mechanism. There are two primary reasons for this:

- First, the TLB flushes are more expensive under the new mechanism. When a guest domain's page is unmapped from the driver domain's address space, the driver domain CPU's TLB has to be flushed. But since it is the guest domain that is initiating this operation, a remote TLB flush has to be performed using inter-processor interrupts (IPIs). So the guest domain issues an IPI and waits until the TLB is flushed by the driver domain. This is an expensive operation. But under the original grant mechanism, the TLB flushes are local since it is the driver domain that also initiates this operation. But this scenario could change depending on the experimental setup. For instance, the driver domain could be configured to run on multiple CPUs. Then when a page is unmapped, the TLBs corresponding to all the CPUs might have to be flushed, which would then necessitate the expensive IPIs.

- Second, it is not feasible to achieve the same degree of batching under the new mechanism as compared with the original grant mechanism. Under the new mechanism, in the transmit side, only grant operations of large TSO packets can be batched. In the receive side, batching depends on the number of buffers to be refilled in the receive I/O ring at any given time.

Due to these reasons the CPU cost increased by $\sim$88% in the TX experiment and by $\sim$14% in the RX experiment. But a disproportionate increase in the CPU cost (by $\sim$475%) was observed in the inter-VM experiment. This was a result of significant

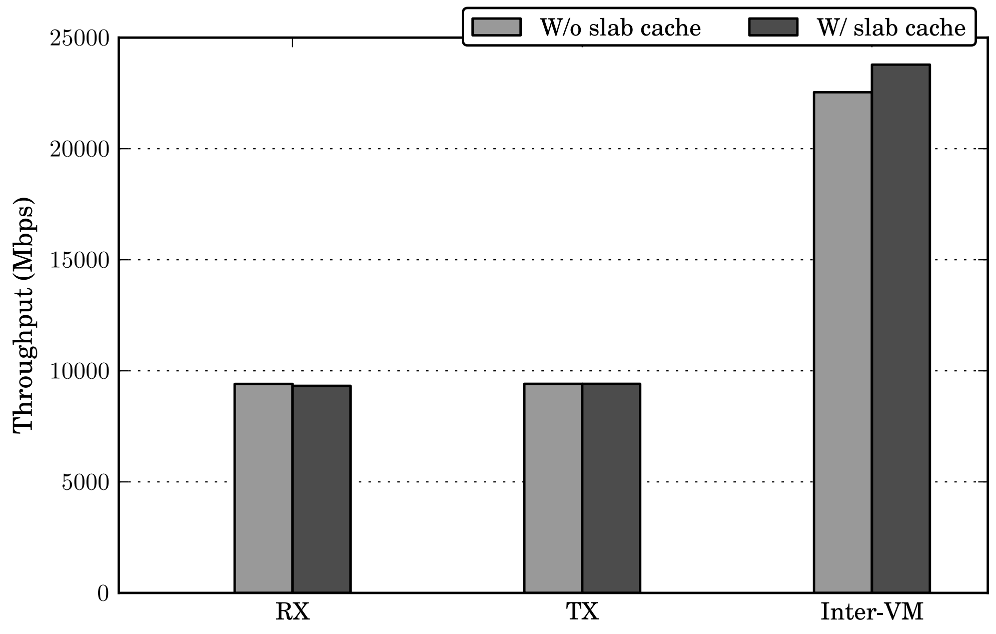


Figure 3.12 : *Effect of Slab Cache - Throughput Results*

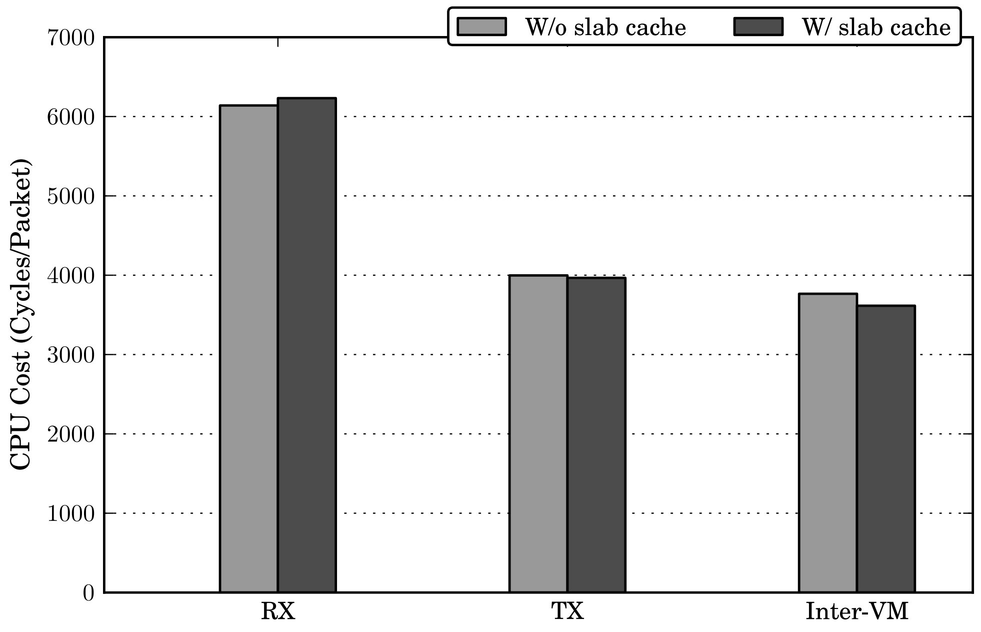


Figure 3.13 : *Effect of Slab Cache - CPU Cost Results*

lock contention inside the hypervisor. A lock is held while manipulating an IOMMU table. Similarly, another lock is held while performing the IOTLB flushes. Now, in the inter-VM experiment, there were two guest domains performing the grant operations concurrently. This means both were attempting to access the same IOMMU table within the hypervisor but such accesses were serialized. So this resulted in a major lock contention which significantly increased the CPU cost. But as mentioned earlier, typically, there is always going to be some temporal/spatial locality in a guest domain's utilization of I/O buffers. So this worst case behavior is not expected.

Figures 3.12 and 3.13 show the effect of using the slab cache to implement the reuse scheme (discussed in Section 3.5.1) on throughput and CPU cost respectively. The use of the slab cache has two opposing effects:

1. The number of revocations goes up due to the periodic draining of unused buffers from the slab cache by the guest OS. This increases the CPU cost.

2. The process of allocating and freeing I/O buffers is made more efficient. This decreases the CPU cost.

So the net effect depends on which of these two effects dominate. In the TX experiment, line rate (∼9400 Mbps) was achieved in both cases, with negligible difference in CPU cost. In the RX experiment, effect (1) dominated, and this resulted in a minor decrease in throughput, from ∼9400 to ∼9300 Mbps. In the inter-VM experiment, effect (2) dominated, which resulted in a minor increase in throughput, from ∼22500 to ∼23700 Mbps. Also, since the locality in the guest domain's utilization of I/O buffers was already quite high, the use of the slab cache did not have any further impact on locality.

3.7 Conclusions

In driver domain-based I/O architectures, memory is shared between guest and driver domains, and with devices. The overheads incurred due to the memory sharing mechanisms often limit the achievable I/O performance. This chapter presented a new memory sharing mechanism that allows a guest VM to take advantage of temporal and/or spatial locality to reuse shared I/O buffers across multiple I/O operations. The key idea in the new mechanism is to allow guest domains to unilaterally revoke grants. This in turn allows a guest VM to delay the grant revocations after the completion of I/O operations to any time in future. Further, the new mechanism also provides a unified interface for controlled memory sharing with driver domains and with I/O devices using the IOMMU hardware.

A prototype of the new mechanism was implemented in the Xen virtualization platform. An evaluation of this prototype showed that it reduced the CPU cost during I/O operations by up to 45% and increased the throughput by up to 150% under Xen. While this work only explored the use of the new mechanism for network I/O, it can completely replace the existing memory mechanism in Xen. In general, the new mechanism is applicable to any driver domain-based I/O architecture and in particular, any memory sharing scenario that exhibits significant temporal and/or spatial locality should benefit from the new mechanism. This chapter has shown that one can reap the benefits of driver domains without needing to sacrifice performance.

Chapter 4

Hyper-Switch - A Scalable Software Virtual Switching Architecture

4.1 Introduction

Today, software device virtualization is far more widely used than specialized hardware devices. This is due in part to the rich set of features—including security, isolation, and mobility—that software-based solutions offer. These solutions can be classified into driver domain and hypervisor-based architectures as detailed in Chapter 2. The virtue of driver domain-based architectures is that they provide a safe execution environment for physical device drivers that are used to access the underlying devices. The hypervisors that support driver domains are more robust and fault tolerant, since they have a much smaller trusted computing base (TCB), as compared with the alternate solutions that locate the device drivers within the hypervisor. However, on the flip side, they incur significant software overheads that not only reduce the achievable I/O performance but also severely limit I/O scalability [34, 41].

The virtualization of network devices is fundamentally different and more complex than virtualizing other I/O devices (e.g., block devices) since it entails a virtual switching component. Packets that are received from a real network device must be first switched to determine the destination VM. Until then the packets must be processed in an as-yet-unknown context. Typically, this last hop virtual switch is also implemented inside the same software domain where the virtual devices are implemented and the device drivers are hosted. For instance, all these components are implemented inside a driver domain in Xen [8] and the hypervisor in KVM [11] and VMware ESX server [10]. This colocation is purely a matter of convenience since network packets must be switched while they are moved between the virtual devices and the device drivers.

This thesis describes the *Hyper-Switch* that challenges these conventional architectures by separating the virtual switch from the domain that hosts the device drivers. Hyper-Switch is a highly streamlined, efficient, and scalable software-based last hop virtual switching architecture, specifically for hypervisors that support driver domains. In particular, the hypervisor includes the data plane of a flow-based software switch, while the driver domain continues to safely host the device drivers. Since the data plane is typically small it does not significantly increase the size of the hypervisor and therefore, the size of the system's trusted computing base (TCB). For instance, the data plane of Open vSwitch [36], a popular software switch, includes ∼5K lines of source code. The control plane of the switch, which is significantly larger (∼90K lines of code in Open vSwitch) resides in the virtualization management layer. So the Hyper-Switch solution explores a new point in the virtual switching design-space.

Another contribution of this paper is a series of optimizations that increase performance. They allow the Hyper-Switch architecture to efficiently support both bulk and latency sensitive network traffic. This includes *VM state-aware batching* of packets to mitigate the cost of hypervisor entries on the transmit side and the cost of guest notifications on receive side. *Preemptive copying* is employed, while a VM is being notified, to reduce packet latency at the receiving VM. Further, whenever possible, the network packet processing is *dynamically offloaded* to idle CPU cores. The offloading is performed using a simple, low-overhead mechanism that is optimized to take advantage of CPU cache locality, especially in NUMA systems.

These optimizations enable efficient packet processing, better utilization of the available CPU resources, and higher concurrency. They take advantage of the Hyper-Switch's integration within the hypervisor and its proximity to the scheduler. So the Hyper-Switch uses the information on when and where a VM is running to optimize packet delivery. As a result, the proposed architecture enables much improved and scalable inter-VM network performance, while still maintaining the robustness and fault tolerance from driver domains. Further, we believe that these optimizations can and should be a part of any virtual switching solution that aims to deliver high performance.

The Hyper-Switch architecture was evaluated using a prototype that was implemented in the Xen virtualization platform [24]. The prototype was built by modifying Open vSwitch [36], a multi-layer software switch for commodity servers. In the evaluation, the Hyper-Switch outperformed both Xen's default network I/O architecture and KVM's vhost-net architecture. For instance, on a 32-core machine, in the pairwise scalability experiments the Hyper-Switch achieved a peak net throughput of ∼81 Gbps as compared to only ∼31 Gbps and ∼47 Gbps under Xen and KVM respectively.

The rest of this chapter is organized as follows. Section 4.2 further motivates the Hyper-Switch architecture by discussing some of the issues with existing solutions. Section 4.3 explains the design of the Hyper-Switch architecture. Section 4.4 describes the implementation the Hyper-Switch prototype. Section 4.5 presents a detailed evaluation of the Hyper-Switch. Section 4.6 discusses ideas to further enhance the Hyper-Switch architecture.

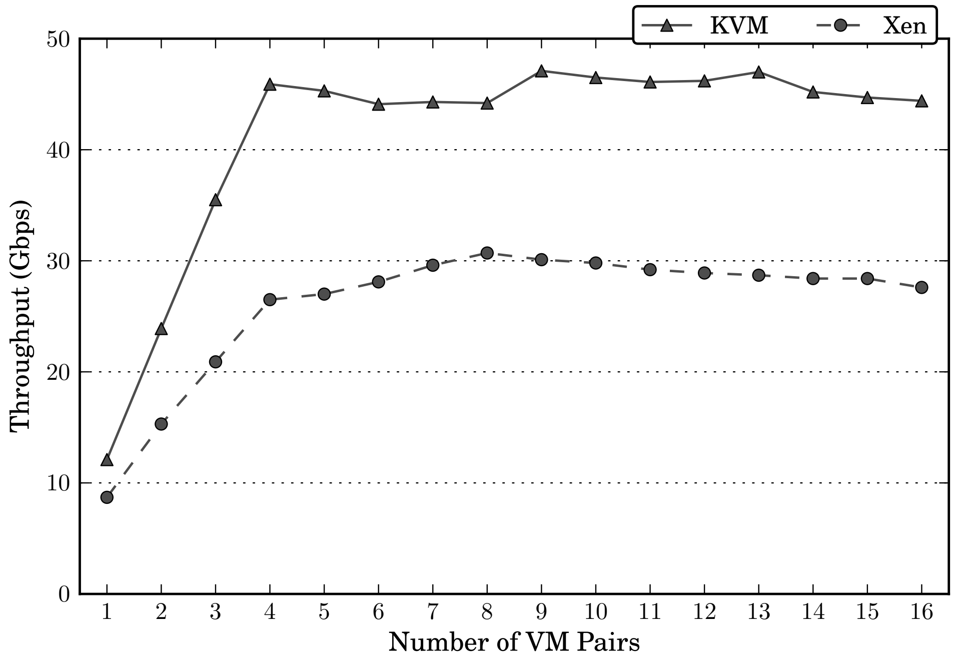


Figure 4.1 : KVM vs Xen: Network Performance

4.2 Motivation

In virtualized servers, the last hop virtual switch plays a critical role in forwarding several types of network traffic—including internal network traffic between virtual machines (VMs) co-located on the same server, and external network traffic from other servers within the datacenter, clients outside the datacenter, and even remote storage traffic. So the virtual switch must be highly efficient and optimized to enable high performance. In order to achieve this efficiency, it must be an integral part of the hypervisor.

Hypervisors that support driver domains are potentially more robust and fault tolerant. However, driver domains incur significant overheads. Previous studies have identified memory sharing as one of the major sources of overhead in using driver domains [34, 41, 47]. This overhead is due to the costs incurred when a guest VM interfaces with the driver domain. This is required to move packets between the guest VMs and the driver domain. Ordinarily, the driver domain cannot directly access a packet in the guest VM's memory since it is just another VM and the hypervisor has to maintain memory isolation between all VMs. So memory has to be explicitly shared between them. For instance, in Xen, this memory sharing is supported using the *grant mechanism*[2]. Our experiments show that the grant operations alone account for nearly 60% of the CPU cycles consumed in the driver domain while processing network packets. Consequently, the driver domain is often a network performance bottleneck. Hypervisor-based architectures do not incur the memory sharing overheads since the packets in the guest VMs' memory can be directly accessed from the hypervisor. Figure 4.1 compares the network performance between Xen and KVM where network traffic was setup between 1–16 pairs of VMs. In both the cases, Open vSwitch was used to switch packets. Clearly, the performance under KVM, which supports the hypervisor-based architecture, scales better. So one of the main design goals in Hyper-Switch was to eliminate the memory sharing overheads due to driver domains.

Further, there is also the cost of setting up and tearing down the IOMMU mappings that are required to allow the I/O device to safely access the network packets in the host memory [40, 49]. In Xen, the overhead due to the IOMMU operations are incurred for *all* network traffic, whether they are internal or external. This is due to the coupling of the IOMMU operations with the grant operations in the driver domain.

In modern multi-core systems, concurrent processing of packets is essential to achieve high performance. Under Xen's default network architecture using driver

[2]Xen's grant mechanism is explained in detail in Chapter 3.

domains, the driver domain can be run alongside the transmitting and receiving VMs. Consequently, it is possible to perform packet switching in parallel with packet transmission and reception. However, until recently, the Xen backend driver (called *netback*) in the driver domain was single-threaded. So it was not possible to scale the packet processing in the driver domain beyond a single CPU core. Since the driver domain handled all network traffic, it was a significant performance bottleneck. In order to address this limitation, recently, netback was upgraded to support multiple threads of execution[3].

Despite this improvement, there are several fundamental problems with traditional driver domain architectures that limit I/O performance scalability. Fundamentally, the driver domains must be scheduled to run whenever packets are waiting to be processed. This might involve scheduling multiple vCPUs depending on the number of threads used for packet processing in the driver domain. As a result, the scheduling overheads are incurred while processing network packets. Further, the driver domain must be scheduled in a timely manner to avoid unpredictable delays in the processing of network packets. There have been several proposals that attempt to optimize the scheduling of driver domains to achieve high performance [48, 50–52]. But, fundamentally, this is a hard problem to solve for all workloads.

Today, it is standard practice in real virtualization deployments to dedicate processor cores to the driver domain. This avoids any scheduling delays. However, the dedicated CPU cores would lie unused when there is no network activity in the system. In fact, dedicating CPU resources for backend processing is not limited to just driver domain-based architectures. There have also been several proposals to offload some of the packet processing to dedicated CPU cores—including Liu *et al.*'s virtualization

[3]Added in May 2010 to the Xen pv-ops domain-0 git development repository.

polling engine (VPE) [53], Kumar *et al.*'s sidecore approach [54], and Landau *et al.*'s split execution (SplitX) model [55]. But, as mentioned earlier, dedicating CPU cores for packet processing can lead to under-utilization of the processor cores. Further, this goes against one of the fundamental tenets of virtualization, which is to enable the most efficient utilization of the server resources. Therefore, another goal in designing the Hyper-Switch architecture was to avoid dedicating resources and instead, to dynamically utilize the available resources.

But packet processing cannot be simply offloaded to any idle core in the system. Modern processors support several levels of CPU cache hierarchy to reduce the gap between processor and memory speeds. Moreover, the CPU caches are often shared between two or more processor cores. When a processor core reads or writes a memory location, it is first brought into its cache. When a second processor core accesses the same memory location, depending on whether the two cores shared the cache or not, the memory access can be fast or slow. If the cores shared the cache, then the second processor can quickly access the data from the shared cache. Otherwise, depending on the processor's cache coherence mechanism, data might have to be written back to the DRAM and then brought into the second processor core's cache. Therefore, cache locality can have a significant impact on packet copying costs. So the CPU cores must be carefully chosen to intelligently exploit any cache locality. This not possible in architectures where CPU cores are statically dedicated for packet processing.

Another challenge in virtualized systems is the proper accounting of the I/O virtualization overheads, which also includes the cost of switching and forwarding packets. This problem is exacerbated when driver domains are used since it is extremely hard to account for the CPU cycles consumed in the driver domain. For instance, Gupta *et al.* [56] explain this problem in Xen and propose an elaborate infrastructure to enforce performance isolation through proper attribution of the resources consumed. So the Hyper-Switch architecture is designed such that the I/O resource accounting is implicit and therefore, significantly simplified.

One can imagine a system having multiple driver domains, each with access to one or more network interfaces. This will provide the hypervisor with more opportunities to isolate different network traffic types from each other and enforce access restrictions to guarantee QoS by controlling when and how long the driver domains are run. For instance, the inter-server synchronization traffic can be forwarded through one driver domain and the storage traffic through a different driver domain. The hypervisor can favor the synchronization traffic by scheduling the associated driver domain at a higher priority than the driver domain used for the storage traffic. In traditional drive domain-based architectures it is hard to support multiple driver domains since the last hop virtual switch is tied to the driver domain. A potential solution might be to build a separate L2 switching domain for each type of traffic. But this would be significantly more complex to implement and support. However, moving the virtual switch out of the driver domain essentially solves this problem.

4.3 Hyper-Switch Design

Figure 4.2 illustrates the Hyper-Switch architecture. There are two fundamental aspects to this architecture. First, unlike existing systems that use driver domains, the Hyper-Switch architecture—as the name implies—implements the virtual switch inside the hypervisor. So internal network traffic between virtual machines (VMs) that are co-located on the same server is handled entirely inside the hypervisor. Incoming external network traffic is initially handled by the driver domain, since it hosts the device drivers, and then is forwarded to the destination guest VM through the Hyper-

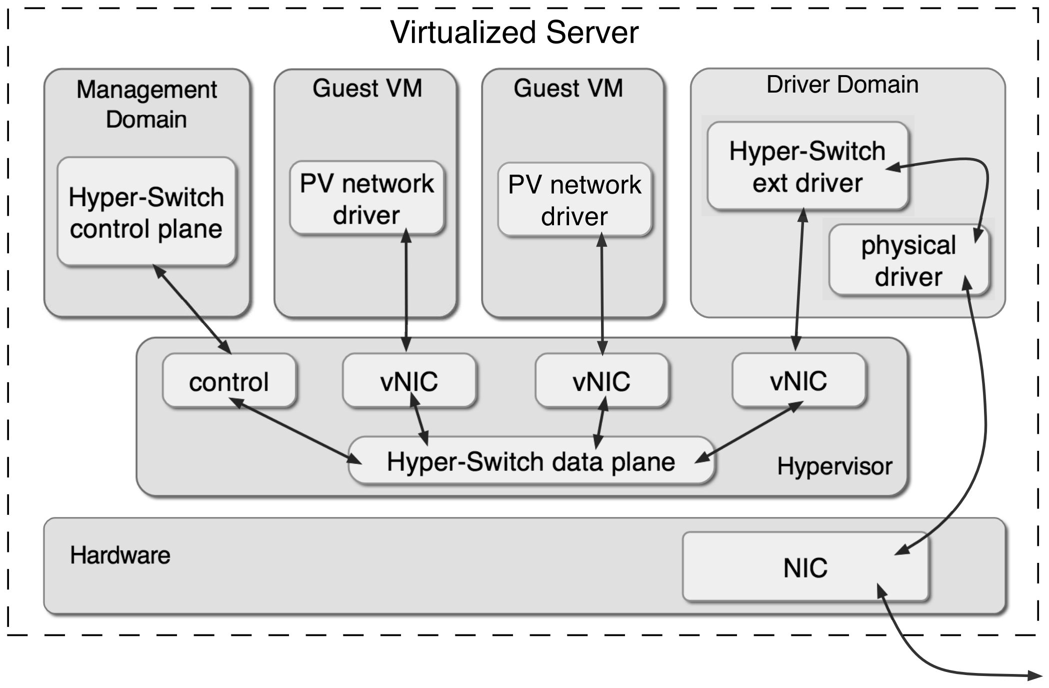


Figure 4.2 : *The Hyper-switch architecture. The last hop virtual switch is implemented partly in the hypervisor (data plane) and partly in the management layer (control plane). The device drivers are hosted in the driver domain.*

Switch. Outgoing external traffic is handled similarly but in the opposite direction. In essence, from the Hyper-Switch's perspective, two guest VMs form the endpoints for internal network traffic, and the driver domain and a guest VM form the endpoints for external network traffic. But in traditional driver domain architectures, all network packets are forwarded to the driver domain first, where they are switched, and then if needed the packets are moved to the I/O device using the device drivers hosted in the driver domain.

Second, the hypervisor implements just the data plane of the virtual switch that is used to forward network packets between VMs. The switch's control plane is implemented in the management layer. So the virtual switch implementation is distributed across virtualization software layers with only the bare essentials implemented inside the hypervisor. The separation of control and data planes is achieved using a flow-based switching approach. This approach has been previously used in other virtual switching solutions such as the Open vSwitch [36]. However, in traditional driver domain-based architectures, Open vSwitch's control and data planes are both implemented inside the driver domain.

In Hyper-Switch, the packets belonging to internal network traffic are delivered directly to the destination guest VM by the hypervisor and therefore, do not incur the memory sharing or IOMMU overheads. This enables highly efficient inter-VM networking. In fact, these overheads are not incurred even for external network traffic since the packets are never directly moved between the guest VMs and the driver domain. Instead they are always forwarded through the Hyper-Switch in the hypervisor. All that the driver domain has to do is send and receive packets using the appropriate device driver.

Since the driver domain is used only for external communication, it does not require significant CPU resources. More precisely, experiments have shown that for sending or receiving network packets at 10 GbE line rate, the driver domain does not require more than a single CPU core. So the problems that result from the use of driver domains is significantly reduced. In particular, the overheads due to driver domain scheduling was entirely avoided at least for inter-VM communication through Hyper-Switch.

The rest of this section describes the Hyper-Switch's design in detail. First, the basic design is explained by describing the path taken by a network packet through Hyper-Switch. This is followed by several optimizations that improve the basic design to enhance performance.

4.3.1 Basic Design

From Hyper-Switch's perspective, the packet processing begins at the transmitting VM where the network packet originates and ends at the receiving VM where the packet has to be delivered[4]. The packet processing occurs in four stages: (1) packet transmission, (2) packet switching, (3) packet copying, and (4) packet reception.

Packet Transmission

In the first stage, the transmitting guest VM pushes the packet to the Hyper-Switch for processing. Packet transmission begins when the network stack in the guest VM forwards the packet to its para-virtualized network driver. Then the packet is queued for transmission by setting up descriptors in the transmit ring. Each descriptor identifies a location in the guest VM's memory where the packet data is present.

A single packet can potentially span multiple descriptors depending on its size. Typically, packets are never segmented in the transmitting guest VM. In other words, segmentation offload is always enabled in a guest VM's virtual network interface. The packets belonging to internal network traffic are never segmented. So packets, up to the size of 64 KB[5], can be forwarded as is. The external packets are segmented either in the driver domain or the network hardware. The latter happens when the hardware is capable of performing segmentation. Today, segmentation offload is a standard feature in most modern network devices.

[4]For external network traffic, the driver domain is either the transmitting or the receiving VM.

[5]The 64 KB packet size limit is due to IP restrictions.

Once a packet has been queued for transmission, the Hyper-Switch has to switch the packet to find its destination. This is triggered by a hypercall issued by the transmitting VM. The guest VM enters the hypervisor and calls the Hyper-Switch routine that inspects the descriptors queued in the transmit ring and reconstructs the packet. Packets are never copied on the transmit side. The Hyper-Switch directly references the data in the transmitting VM's memory.

Packet Switching

In the second stage of packet processing, the packet is switched to establish its destination. Packet switching begins when the packet is removed from the transmit ring and is pushed to the Hyper-Switch's data plane where it is processed using flow-based packet switching approach. The Hyper-Switch must be able to read the packet headers to switch it. But since the packet is not copied on the transmit side, the packet headers are read directly from the transmitting VM's memory. This is feasible since the hypervisor has direct access to all guest VMs' memory.

A packet is switched in the Hyper-Switch's data plane in three steps:

1. **Flow identification:** The packet header fields are parsed to identify the corresponding packet flow.

2. **Flow table lookup:** The packet flow, identified in the previous step, is used to lookup a matching flow rule in a software flow table. When the flow table lookup fails, *i.e.,* a cache miss, the packet is forwarded to the control plane in the management layer.

3. **Flow action execution:** A successful flow table lookup, *i.e.,* a cache hit, identifies a flow rule, which specifies one or more actions to be performed.

Typically, the action is to forward the packet to one or more ports or to drop the packet. Each output port identifies a virtual network interface within the destination guest VM.

When the flow table lookup fails, the packet is forwarded to the control plane through a separate *control interface*. The control plane decides how the packet must be forwarded based on packet filtering rules, forwarding entries from an Ethernet address learning service, and/or other protocol specific tables. This is composed into a new flow rule that specifies the actions to be performed on packets belonging to this flow. Then the packet is re-injected into the Hyper-Switch's data plane and the associated actions are executed. Finally, the control plane adds the new flow rule to the flow table. This allows the flow's subsequent packets to be handled entirely within the Hyper-Switch's data plane.

Once the switching is completed, the destination ports are then known. Each destination port has an internal receive queue where the switched packet is temporarily placed.

Packet Copying

In the third stage of packet processing, the switched packet is copied into the receiving VM's memory. Empty receive buffers are provided in advance at a virtual network interface for receiving new packets. This is done by setting up descriptors in the receive ring that provide the address of the empty buffers in the guest VM's memory.

In the proposed architecture, by default, the destination VM is responsible for performing the packet copies. Once switching is completed, the destination VM is notified via a virtual interrupt. Subsequently, that guest VM issues a hypercall to enter the hypervisor. While in the hypervisor, it dequeues the packet from its internal receive queue, and copies the packet into the memory referenced by the next descriptor in its receive ring. The packet is copied directly from the transmitting VM's memory to the receiving VM's memory from the hypervisor.

After the packet has been copied, the memory that was allocated for this packet at various places is released. First, this happens inside the hypervisor, where the data structures used to construct the packet are freed. Then the transmitting VM is notified so that it can release both the packet data structures and the buffers that held the packet contents.

So, by default, the packet processing in the Hyper-Switch's data plane is performed in a specific guest VM's context. Packet switching is the responsibility of the transmitting VM. So it happens only in its context. Packet copying is performed only in the receiving VM's context. So this solves the resource accounting problems detailed in Section 4.2.

Packet Reception

In the fourth and final stage, the para-virtualized network driver in the receiving guest VM reconstructs the packet from the descriptors in the receive ring. Typically, the receiving OS is notified, through interrupts, that there are new packets to be processed in the receive ring. But this is not necessary here since the receiving guest VM is already notified in the previous stage. So the packet reception can happen as soon as the hypercall that was issued to copy the packet is completed. The new packet is then pushed into the receiving VM's network stack.

4.3.2 Preemptive Packet Copying

As mentioned before, packet copies are performed, by default, in a receiving VM's context. When a packet is placed in the internal receive queue, after it has been switched, the receiving VM is notified. Eventually, the receiving VM enters the hypervisor to copy the packet. But the notification of a VM also requires hypervisor intervention. For instance, under Xen, when there is a pending notification to a VM, the VM is interrupted and pulled inside the hypervisor. Then the hypervisor has the guest VM handle all the pending notifications.

Hypercalls (like system calls) are expensive operations since they involve context switching to the hypervisor that is running in a privileged mode. Therefore, the overhead of issuing the hypercall and entering the hypervisor for copying is eliminated by *preemptively* copying the packet when the receiving guest VM is being notified. In essence, the packet copy operation is combined with the notification of the receiving VM. This optimization avoids one hypervisor entry for every packet that is delivered to a VM. This shows the advantage of deeply integrating the switch with the functioning of the hypervisor.

4.3.3 Batching Hypervisor Entries

In the Hyper-Switch architecture, as described thus far, the transmitting VM enters the hypervisor every time there is a packet to send. Moreover, the receiving VM is notified every time there is a packet pending in the internal receive queue. As mentioned earlier, even this notification requires hypervisor intervention[6]. Therefore,

[6]In Xen, notifying a running guest VM involves two entries into the hypervisor. First, the running VM is interrupted via an IPI and forced to enter the hypervisor. Then the hypervisor runs a special exception context where the guest VM handles all pending notifications. Finally, the guest VM

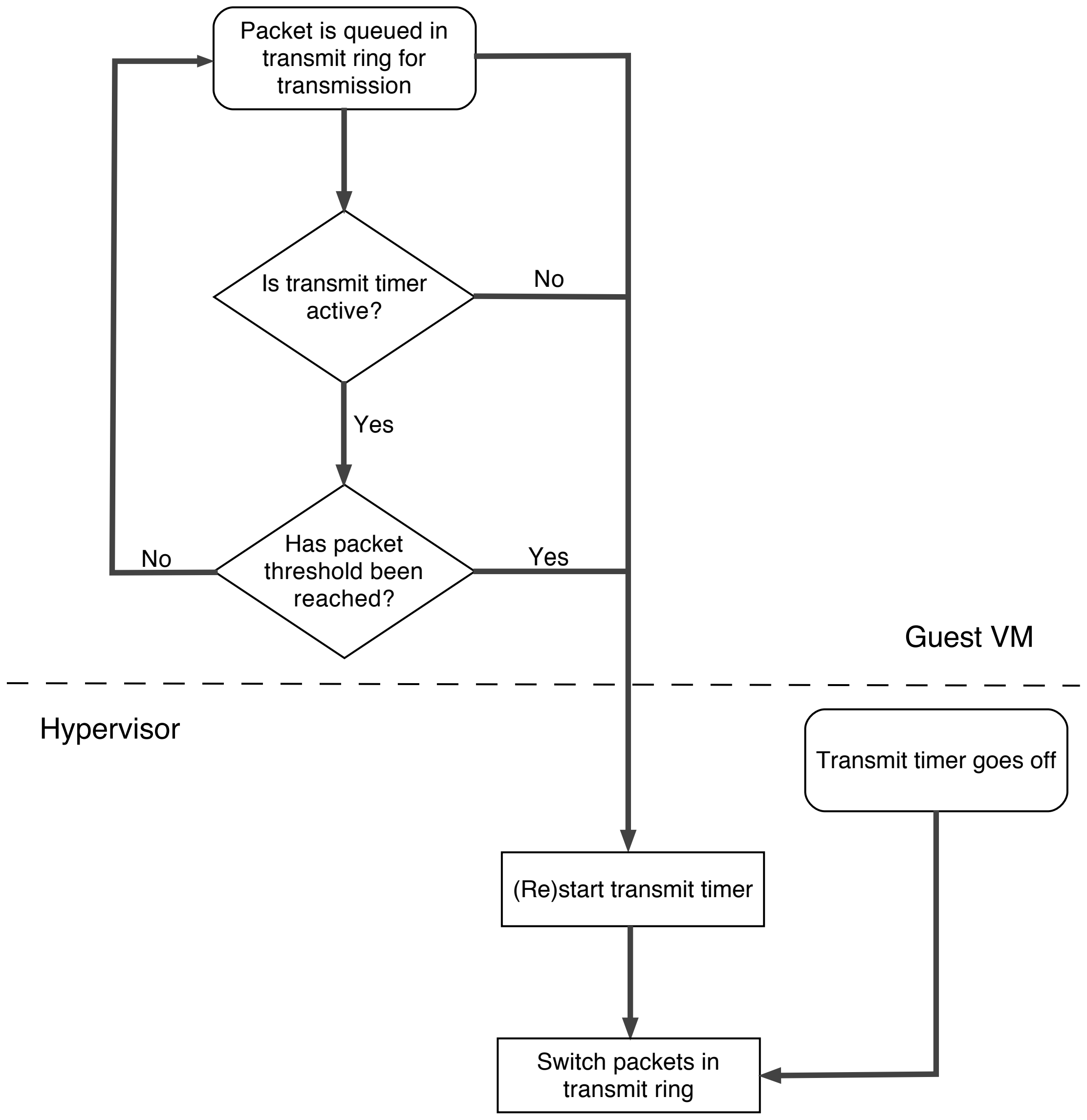

Packet is queued in transmit ring for transmission
Is transmit timer active?
No
Yes
Has packet threshold been reached?
No
Yes
Guest VM
Hypervisor
Transmit timer goes off
(Re)start transmit timer
Switch packets in transmit ring

Figure 4.3 : *Flow chart depicting the working of transmit timers. The timers are used to batch the entries from the transmitting VM into the hypervisor to switch packets.*

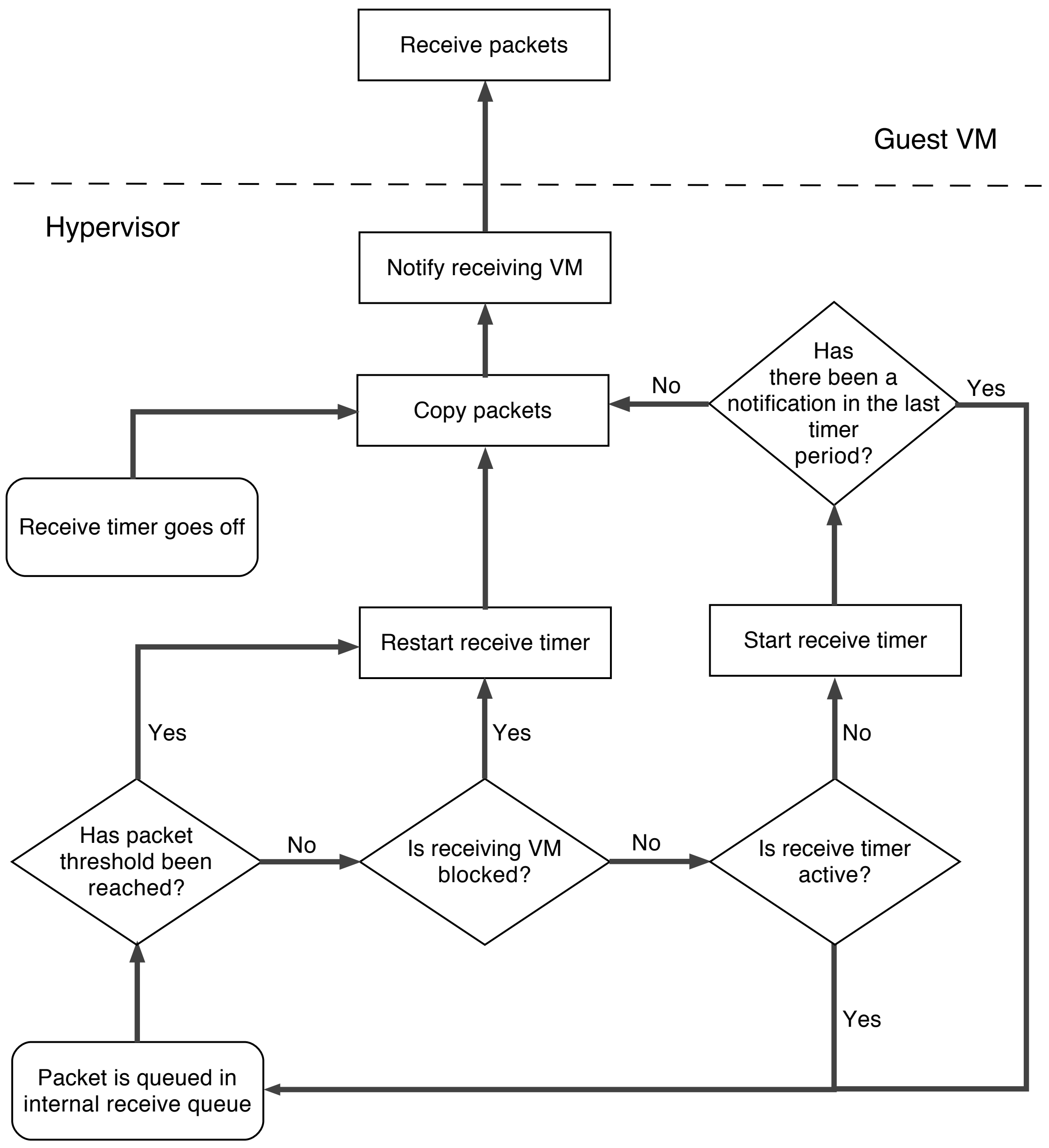


Figure 4.4 : *Flow chart depicting the working of receive timers. The timers are used to batch the notifications to the receiving VM.*

despite the use of the preemptive packet copy optimization, the overhead of entering of the hypervisor is incurred **multiple times** on **every packet**. This is simply not compatible with achieving high performance.

To mitigate this overhead, *VM state-aware batching* is used, which amortizes the cost of entering the hypervisor across several packets. This approach to batching shares some features with the interrupt coalescing mechanisms of modern network devices. Typically, in network devices, the interrupts are coalesced irrespective of whether the host processor is busy or not. But, unlike those devices the Hyper-Switch is integrated within the hypervisor, where it can easily access the scheduler to determine when and where a VM is running. So a blocked VM can be notified immediately when there are packets pending to be received by that VM. This enables the VM to wake up and process the new packets without delay. On the other hand, the notification to a running VM may be delayed if it was recently interrupted.

Timers are used to implement batching. Each virtual network interface has two timers: a *transmit timer* and a *receive timer*. Further, the batching is driven by two parameters: a *timer period* and a *packet threshold*. The timer period determines the duration for which packets are batched. But if the packet threshold is reached before the timer period has elapsed, then the packets are processed immediately. The rationale is that if sufficient packets are batched before the timer period elapses, then there is no benefit from delaying the packet processing until the end of the timer period.

The transmit timer is started just before the first packet is switched. Subsequent packets are queued by the transmitting VM in the transmit ring without entering the hypervisor to switch them. If the packet threshold is reached, then the transmitting

again enters the hypervisor to return from the exception context.

VM enters the hypervisor to switch the packets. Otherwise, the transmit timer goes off at the end of the timer period and all the packets pending in the transmit ring are switched. The flow chart in Figure 4.3 depicts the working of the transmit timer.

If the packet threshold is reached before the timer period ends, then the transmit timer is restarted. Otherwise, the timer is started again only when the next packet is ready for transmission. The rationale behind this scheme is to avoid unnecessary handling of timer events. When the packet threshold is reached, it is inferred that there is a sudden burst of network activity, and that it is highly probable that there are more packets to follow. So the timer is started right away to avoid entering the hypervisor again on the next packet. But when the timer goes off before the packet threshold is reached, it is inferred that there is not a lot of network activity, and hence the timer is started only on the next packet transmission. However, if there is no network activity, then the transmit timer is never started again.

The receive timer is implemented similarly. The receive timer is started when the first packet is queued for copying at a virtual network interface's internal receive queue. When the receive timer goes off after the timer period, the receiving VM is notified. Then the packets pending in the internal receive queue are preemptively copied into the receiving VM's memory.

Typically, the notification to the receiving VM is delayed until the end of the current receive timer period. However, there are three exceptions. The receiving VM is notified immediately after a packet is queued when:

- The packet threshold is reached. Then it is assumed that there are enough packets pending in the internal receive queue to warrant immediate copying.
- The receiving VM is blocked and not currently running. Then it is considered

worth incurring the overhead of notifying the receiving VM to wake it up and have it process the new packet.

- The receive timer is off and there has not been a notification in the last timer period. In other words, the first packet to be queued before the timer is started results in a notification, but only if there hasn't been a notification in the last receive timer period. This avoids the situation where the receiving VM is sent back-to-back notifications when a packet is queued right after the timer goes off.

The timer is restarted in the first two cases so that subsequent packets continue to be queued. The flow chart in Figure 4.4 depicts the working of the receive timer.

The heuristics used in implementing the timers ensure immediate processing of packets that arrive after a period of inactivity at a virtual network interface. This can be beneficial for latency sensitive request/response type of traffic since the packets are handled promptly without any delay.

4.3.4 Offloading Packet Processing

In Hyper-Switch, by default, packet switching is performed in the transmitting VM's context and packet copying is performed in the receiving VM's context. As a result, asynchronous packet switching does not occur with respect to the transmitting VM and similarly, asynchronous packet copying does not occur with respect to the receiving VM. However, concurrent and asynchronous packet processing can significantly improve performance.

Concurrent packet processing is supported by polling: (1) all the internal receive queues, looking for packets waiting to be copied and (2) all the transmit rings, looking

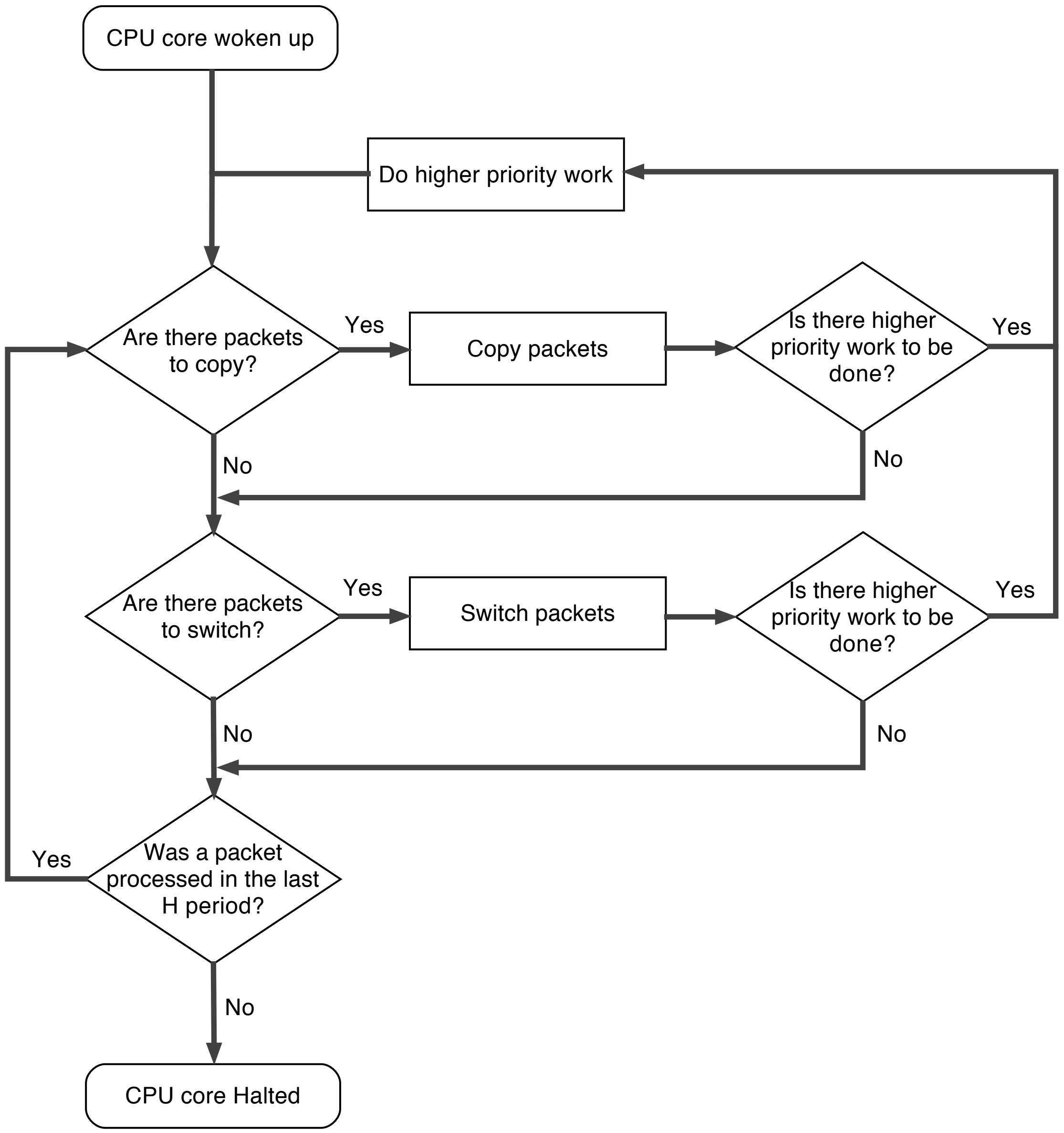

CPU core woken up
Do higher priority work
Are there packets to copy?
Yes
Copy packets
Is there higher priority work to be done?
Yes
No
No
Are there packets to switch?
Yes
Switch packets
Is there higher priority work to be done?
Yes
No
No
Yes
Was a packet processed in the last H period?
No
CPU core Halted

Figure 4.5 : *Flow chart depicting the packet processing on idle cores (H period = the hysteresis time period).*

for packets waiting to be switched. This can be performed from the processor cores that are currently idle in the system. Packet copying is prioritized over switching because packet copying is typically the more expensive operation and the receiving VM is more likely to be performance bottlenecked than a transmitting VM. While polling all the virtual network interfaces from all the idle cores in the system might enable higher performance, it is extremely costly in terms of power consumption. On the other hand, if the idle cores halt after a few rounds of polling, then they might not be ready to promptly copy or switch packets at a later time.

Instead, in the Hyper-Switch, the idle cores are woken up just when there is work to be done. On the receive side, this can be ascertained precisely when packets are placed in an internal receive queue of a virtual network interface. Then one of the idle cores in the system is chosen and woken up to perform the packet copy. A simple, low-overhead mechanism is used to offload work to the idle cores. It uses a lightweight inter-processor messaging facility to request a specific idle core to copy packets at a specific virtual network interface. Further, this mechanism attempts to spread the work across many idle cores in the system. Otherwise, if all the work is offloaded to a single idle core, it might become a performance bottleneck.

The offloading to idle cores is delayed if the receiving VM is going to be notified immediately. As explained previously, this typically happens when the receiving VM is not running. Subsequently, the receiving VM copies a bounded number of packets sufficient to keep it busy, and then if packets are still pending in the internal receive queue, the remaining copies are offloaded to an idle core. The rationale is to immediately copy some packets so that the receiver can start processing them, while the remaining packets are concurrently copied at an idle core.

Unfortunately, it is not as easy to offload packet switching to idle cores. Remember that, in the common case, packets are queued by the transmitting VM in the transmit ring without entering the hypervisor. So it is not possible to offload the switching tasks precisely when packets are queued by the transmitting VM. Therefore, packet switching is performed at the idle cores only as a *side effect* of offloading packet copies. In other words, when an idle core is woken up to perform packet copies, it also polls all the transmit rings looking for packets pending to be switched.

Further, when packets are being processed by an idle core, the Hyper-Switch checks for any other work that might need that core. If so, it aborts the Hyper-Switch related packet processing. This ensures that the offloaded packet processing happens at the lowest possible priority and does not prevent other tasks from using that processor.

CPU Cache Awareness

CPU cache locality can have a significant impact on the cost of packet copying under Hyper-Switch. Essentially, the packet data is accessed in three places[7]: (1) The transmitting guest VM, (2) the packet copier, and (3) the receiving guest VM. The transmitting VM writes the packet data. Then the packet copier reads the data from the transmitting VM's memory and writes it into the receiving VM's memory. Finally, the receiving VM reads the packet data. So the packet data can be potentially brought into three CPU caches depending on the system's cache hierarchy and where the two VMs and the copier are run.

If the receiving VM is also the packet copier, then the packet data is brought into the receiving VM's CPU cache while the copy is performed. Subsequently, when the packet is accessed in the receiving VM, it can be read with low latency from the cache.

[7]Packet switching is ignored here since it only accesses the packet headers.

But if the packet copier runs on an idle core, then the access latency will depend on whether the idle core shares any CPU cache with the receiving VM's processor core or not. Therefore, when an idle core is chosen for offloading, preference is given to those cores which share their CPU cache with the receiving VM.

But, on the downside, sharing a CPU cache can also lead to interference effects. When a packet is copied on the idle core, both the source and destination buffers have to be brought into the cache. Let's say another core that shares this cache is running a VM. Then it is possible that, due to the contention for the shared cache, the data is often evicted from the cache, which worsens the access latency. Further, if the packet being copied on the idle core do not belong to that VM, then its performance is unfairly affected.

Therefore, under the Hyper-Switch, the offload mechanism for packet processing is optimized to take advantage of any CPU cache locality. At the same time, it ensures that the offloaded work does not unfairly affect the performance of other VMs running on cores that share their CPU cache with the idle cores.

Hysteresis Period

Waking up an idle core takes a non-trivial amount of time, particularly when the idle core is using deeper sleep states to save power. Further, the inter-processor interrupts (IPIs) that are used to wake up cores are not cheap. Therefore, a small hysteresis period is introduced to ensure that the idle cores stay awake longer than they normally would. The idea is to keep the cores running, after they are woken up, until there is a period—the hysteresis time period—during which no packets are processed. In other words, the idle cores are kept running as long as there is a steady stream of packets to process (as a result of offloading). The flow chart in Figure 4.5

describes how packets are processed on idle cores.

4.3.5 More Packet Processing Opportunities

A packet that is queued in the transmit ring at a virtual network interface will be *eventually* switched by either the transmitting VM or an idle core. This might happen immediately if some idle core polls this interface looking for packets pending to be switched or it might happen only when the transmit timer period elapses. Therefore, packet switching can potentially be delayed[8].

Consider a guest VM that queues some packets for transmission at its virtual network interface and then blocks. Let's assume that there are no other idle cores in the system. If another VM is scheduled to run on this core, then the queued packets are not going to be switched until the blocked VM is scheduled to run again. But this might happen only at the end of the transmit timer period. Even if the processor core becomes idle after the guest VM blocks, there is no guarantee that the blocked guest VM's packets will be switched at that idle core. In fact, the idle core can end up copying packets destined for other VMs. In essence, a guest VM can block despite its packets waiting to be switched.

When a VM's vCPU blocks, it has to enter the hypervisor to give up its processor. Since the VM is already inside the hypervisor, it might as well as check if there are packets pending to be switched or copied. This allows any packet processing work to be completed before the VM stops running. Also, new packet copies result in a notification to the guest VM. Consequently, instead of blocking, the guest VM returns to process the packets that were just received.

[8]The maximum delay is bounded by the transmit timer period.

4.4 Implementation Details

A prototype of the Hyper-Switch architecture was implemented in the Xen virtualization platform. An existing software switch—Open vSwitch—was used to enable quick prototyping. The Hyper-Switch's data plane was implemented by porting parts of Open vSwitch to the Xen hypervisor. But Open vSwitch's control plane was used without any modification. A new para-virtualized network interface was developed for the guest VMs to communicate with the data plane. The same interface was also used by the driver domain to forward external network traffic. The rest of this section describes each part of the Hyper-Switch prototype in detail.

4.4.1 Open vSwitch Overview

Open vSwitch [36] is an OpenFlow compatible, multi-layer software switch for commodity servers. The control and data planes are separated in Open vSwitch. While the data plane is implemented inside the OS kernel, the control plane is implemented in user space. It uses the flow-based approach for switching packets in its data plane. A two way communication channel between the control and data planes is implemented using the netlink socket interface. The communication is based on a custom datapath protocol. This protocol is used to forward packets between the control and data planes. It is also used by the control plane to issue commands that manipulate the flow rules in the data plane.

In a typical deployment of Open vSwitch as a last hop virtual switch, it is implemented entirely inside a driver domain (Xen) or the hypervisor (KVM). Figure 4.6 depicts a deployment of Open vSwitch inside a driver domain. In the common case, the network traffic between the guest VMs is directly switched by the in-kernel datapath (Open vSwitch's data plane). There can be more than one datapath in the

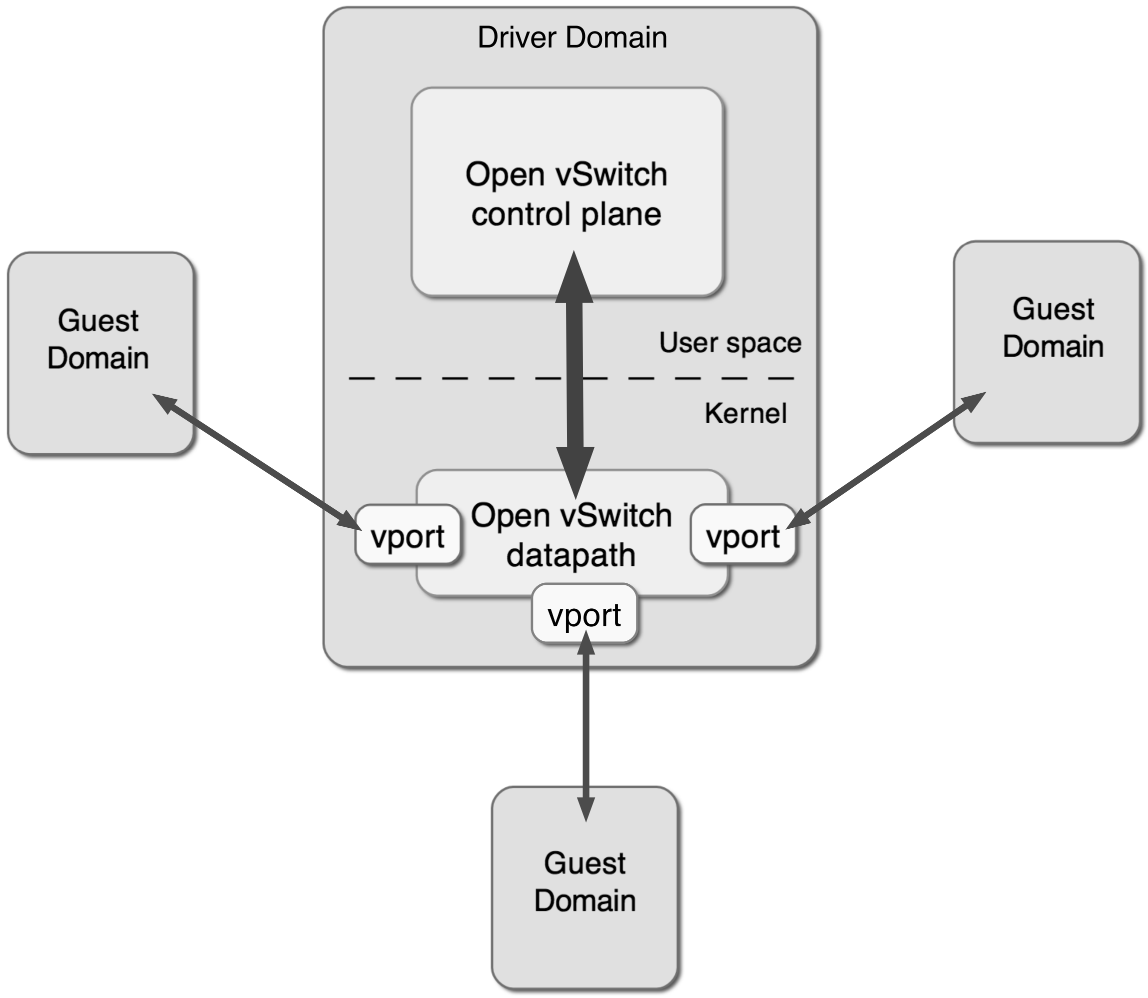


Figure 4.6 : *Open vSwitch as a last hop virtual switch. It is implemented within the driver domain. The in-kernel datapath forwards traffic between VMs. The control plane is implemented in user space.*

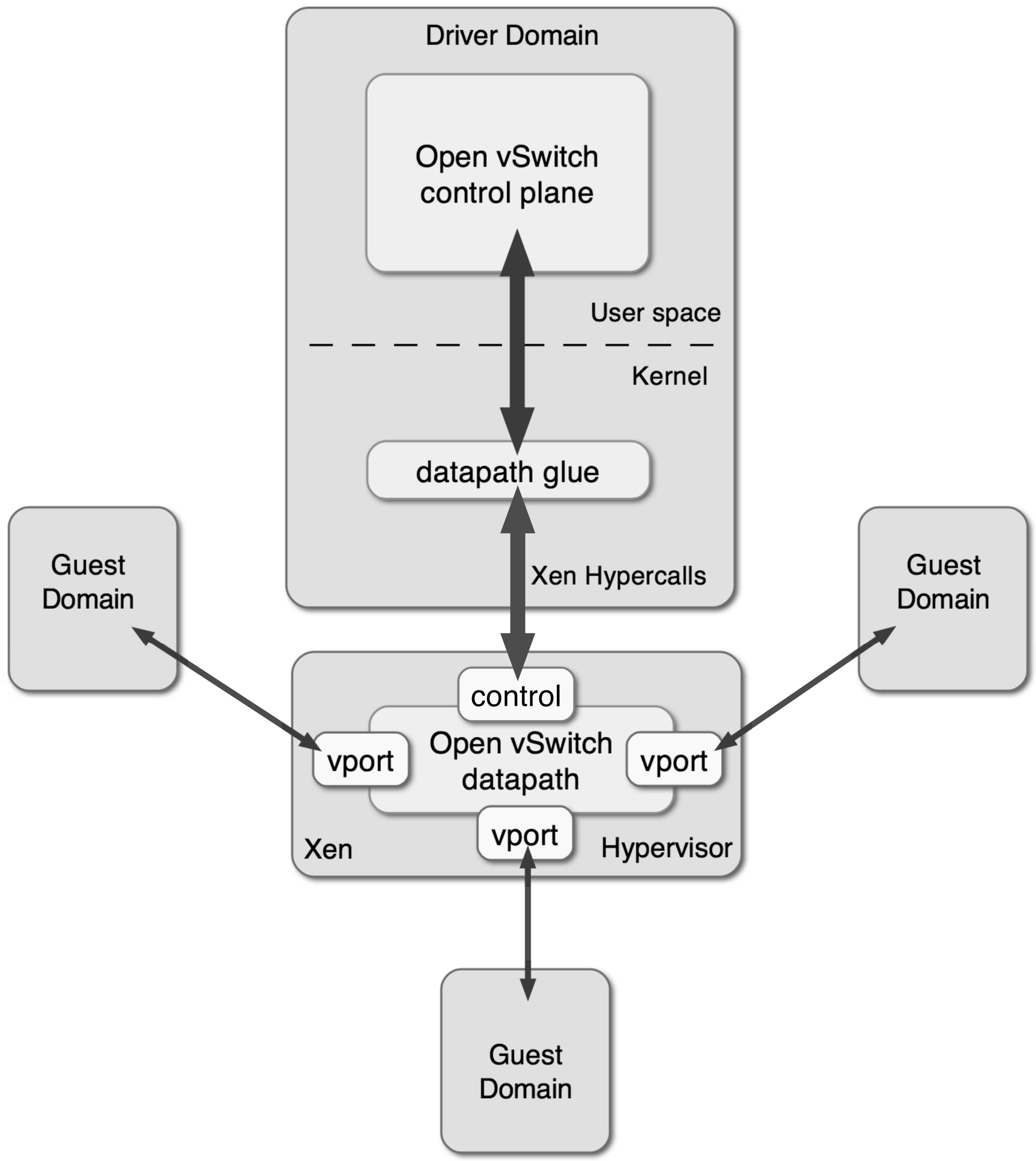


Figure 4.7 : *Hyper-Switch prototype. It was built by porting essential parts of Open vSwitch's datapath to the Xen hypervisor.*

kernel. Open vSwitch provides a *vport* abstraction that can be bound to any network interface in the driver domain. Each vport is attached is one of the datapaths in the system. Typically, there is one vport for every guest VM in the system[9]. The datapath also includes a large software flow table implemented as a hash table.

4.4.2 Porting Open vSwitch's Datapath

The Hyper-Switch's data plane was implemented by porting Open vSwitch's Linux in-kernel datapath to the Xen hypervisor. Just the essential parts of the datapath was ported for the purposes of building this prototype. Specifically, this included the flow table implementation using hash table and the flow table lookup logic. Some Linux kernel libraries—such as the socket buffer interface used to represent the network packets and the flex arrays used to build the hash table—were also ported to Xen. The prototype only supported a single datapath. The vports on the datapath were bound to a newly developed para-virtualized network interface that allowed guest VMs to communicate with the Hyper-Switch's data plane.

A datapath glue layer was implemented in the driver domain kernel to enable communication between the control and data planes. Recall that Open vSwitch's control plane was used without any modification. So it continued to use the netlink socket interface to issue commands and forward packets. The datapath glue layer converted the commands from Open vSwitch's control plan into a new set of Xen hypercalls to manipulate the flow tables in the datapath. The glue layer also transferred the packets that are punted to the control plane. Figure 4.7 depicts the Hyper-Switch prototype.

[9]If the guest VMs have been configured with more than one virtual network device, then there are multiple vports per VM forwarding traffic.

4.4.3 Para-virtualized Network Interface

The guest VMs and the driver domain communicated with the Hyper-Switch through a para-virtualized network interface (vNIC). The interface included two transmit rings—one for queueing packets for transmission and another for receiving transmission completion notifications—and one receive ring to deliver incoming packets. The rings were implemented as fixed circular buffers where the producer and consumer(s) could access the ring descriptors without explicit synchronization. But since the rings could be concurrently accessed from the Hyper-Switch, try-locks were used for synchronization. This was especially beneficial when the rings were polled from idle cores. If the try-lock could not be acquired, then it was simply assumed that someone was else processing the packets from that ring. The interface also included an internal receive queue that contained packets that were yet to be copied into the receiving VM's memory.

4.4.4 Hypervisor Integration

In this architecture, the packet processing is integrated with the functioning of the hypervisor. As explained in Section 4.3.2, packet copying was preemptively performed by combining it with the notification of the receiving guest VM. This was implemented by checking for packets to copy when the associated virtual interrupt was being delivered by the Xen hypervisor to a guest VM. So the guest VMs had to never explicitly enter the hypervisor just to copy packets. Further, packet switching and copying were also performed when a guest VM *voluntarily* blocked. Thus the guest VM's virtual network interface was polled for packets pending to be copied or switched, just before the scheduler was invoked to yield the processor and find another VM to run.

4.4.5 Offloading Packet Processing

The offloading of packet processing was implemented inside Xen's *idle domain*. The idle domain contains one *idle vCPU* for every physical CPU core in the system. The idle vCPUs have the lowest priority among all the vCPUs in the system and therefore, they are scheduled to run on a physical CPU core only when none of the VMs' vCPUs are runnable on that core. The idle vCPUs execute an *idle loop* that checks for pending softirqs and tasklets, and executes the corresponding handlers. Further, the scheduler is also run in a softirq context. Finally, when there is no more work to be done, the processor core enters one of the sleep states to save power.

In the Hyper-Switch architecture, Xen's idle loop was extended to copy and switch packets. Recall that packet copying was offloaded only to specific process cores in the system. So a simple, low-overhead mechanism was used to offload packet processing to idle cores. The mechanism identified a suitable idle core based on an *offload criteria*. The criteria was designed to select an idle core that made the best use of the CPU caches. This is explained in detail in the evaluation section. Further, this mechanism also ensured that the offloaded work was distributed across multiple idle cores using a simple hash function.

The mechanism included a lightweight inter-processor messaging facility that was implemented using small fixed circular buffers. There was one buffer for every processor core in the system. It was used to communicate the vNICs that were being offloaded to a specific idle core. Once the copying was completed, as a side effect, all the transmit rings were also polled looking for packets pending to be switched.

The Hyper-Switch related packet processing was performed only at the lowest priority. The pending softirqs and tasklets were checked after each packet was processed. If there was ever a higher priority work to be done, then the offloaded packet

Table 4.1 : *Hyper-Switch Prototype Parameters*

Parameter	**Value**
Transmit timer period	125 microsecs
Transmit packet threshold	64 packets
Receive timer period	125 microsecs
Receive packet threshold	64 packets
Hysteresis time period	5 microsecs

processing was temporarily aborted. In particular, this was important to ensure that the scheduler was run without any delay.

4.5 Evaluation

This section presents a detailed evaluation of the Hyper-Switch architecture. The evaluation was performed using the Hyper-Switch prototype that was implemented in the Xen virtualization platform. The prototype used para-virtualized (PV) Linux kernels for both the driver domain and the guest VMs. Table 4.1 lists the various configurable parameters and their values as used in this evaluation of the Hyper-Switch prototype. These parameters were carefully tuned to maximize performance.

The Hyper-Switch architecture combines the best of the existing architectures by implementing (part of) the virtual switch inside the hypervisor and hosting the device drivers in the driver domain. The primary goal of this evaluation was to compare Hyper-Switch to the existing architectures that implemented the virtual switch either entirely within the driver domain or entirely within the hypervisor. Toward this end, the Hyper-Switch's performance was compared with that of Xen's default driver domain-based architecture and KVM's hypervisor-based architecture. The evaluation showed that the Hyper-Switch's performance was superior both in terms of the absolute bandwidth and the scalability as the number of VMs and traffic flows were varied.

The Hyper-Switch's external performance was also evaluated. Recall that, under Hyper-Switch, the packets belonging to external network traffic traversed both the driver domain and the hypervisor. But then the packets are not pushed through the network stack in the driver domain. Instead, packets are merely transferred to/from the physical device drivers. Therefore, the packet processing overheads within the driver domain are minimal. This evaluation confirmed the hypothesis that the Hyper-Switch's performance was comparable, despite the seemingly longer route taken by external packets, to the performance under architectures where the packets traversed either only the driver domain or only the hypervisor. Finally, some of the design choices in the Hyper-Switch architecture were also evaluated.

4.5.1 Experimental Setup and Methodology

The experiments were run on an AMD server with two 2.2 GHz Opteron 6274 processors with 16 cores each and 64 GB of memory. So the system had a total of 32 processor cores. The server ran Xen. It was configured to run up to 32 PV Linux guest VMs, and one PV Linux driver domain in addition to the privileged management domain 0. The guest VMs were each configured with a single virtual CPU (vCPU) and 1 GB of memory. The driver domain was configured with up to 8 vCPUs and 2 GB of memory. But under Hyper-Switch, the driver domain was given only a single vCPU since it only handled external network traffic. Table 4.2 summarizes the server configuration.

Table 4.2 : *Hyper-Switch evaluation - server configuration*

H/W configuration		Processor(s)	2 2.2 GHz AMD Opteron 6274
		Total processor cores	32
		Total memory	64 GB
S/W configuration	Hypervisor		Xen 4.2[*]
	Domain 0	Operating System	Linux pv-ops kernel v3.4.4
		Number of vCPUs	1
	Driver domain	Operating System	Linux pv-ops kernel v3.4.4
		Number of vCPUs	1-8
		Memory	2 GB
	Guest VM(s)	Operating System	Linux pv-ops kernel v2.6.38
		Number of vCPUs	1
		Memory	1 GB

[*] Obtained from the Xen hypervisor mainstream git repository (`xen-unstable.git`), as of May 2012.

Table 4.3 : *Hyper-Switch evaluation - client configuration*

Hardware configuration	Processor	2.67 GHz Intel Xeon W3520
	Total processor cores	4 (w/o hyperthreading)
	Total memory	6 GB
Software configuration	Operating System	Linux kernel v2.6.32

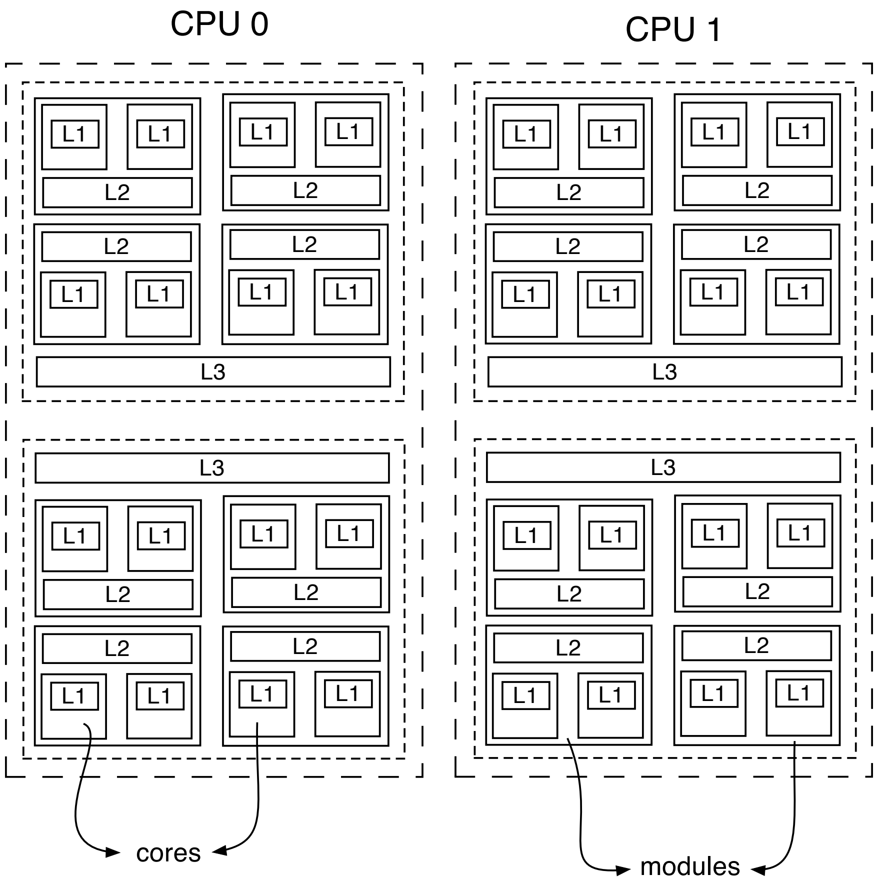


Figure 4.8 : *Hyper-Switch evaluation - server CPU cache hierarchy. L2 cache is shared by 2 cores within a module. L3 cache is shared by 8 cores within 4 modules.*

Figure 4.8 depicts the CPU cache hierarchy in the AMD server. As mentioned earlier, the cache hierarchy has a significant impact on network performance. In particular, transferring packets between source and destination buffers can be more or less expensive depending on where the buffers are cached in the hierarchy. Further, the Hyper-Switch architecture has been optimized to make the best use of a given CPU cache hierarchy. The Opteron processor in the server used in this evaluation was based on AMD's Bulldozer micro-architecture. In this system, each processor core had a private L1 data cache of size 16 KB. Two of the processor cores formed a *module*. The processor cores within a module shared a L2 cache of size 2 MB. Four such modules were placed within a single processor die. Eight processor cores within a die shared a L3 cache of size 8 MB. These cores also formed a processor *node*. So all the cores within a processor node shared the last level cache. Two such dies were packaged in a single processor. There were two such processors in the system. Further, the processor cores within a module also shared an instruction cache (L1i) of size 64 KB (not shown in the figure). The L2 and L3 caches were unified as they held both data and instructions.

The server was directly connected to an external client using a 10 Gigabit Ethernet link. The external client was configured with a 2.67 GHz Intel Xeon W3520 quad-core CPU and 6 GB of memory. It ran an Ubuntu distribution of native Linux kernel v2.6.32. Also, the CPUs at the external client were never a performance bottleneck in any of the experiments. Table 4.3 summarizes the external client configuration.

The Netperf microbenchmark [45] was used in all the experiments to generate network traffic. Netperf is an extremely lightweight benchmark that can be used to generate different types of network traffic. In this evaluation, netperf was used to create two types of network traffic: (1) TCP stream and (2) TCP request/response traffic. The TCP stream traffic was used to measure the achievable throughput. The TCP request/response traffic was used to measure the packet processing latency under different architectures. Unless otherwise specified, the `sendfile` option was used on the transmit side in all the experiments. The experiments primarily evaluated the scenario where the network traffic was setup between the guest VMs co-located on a single server. The evaluation also included some experiments with network traffic from or to the external client.

The performance of Hyper-Switch was compared with the performance of Open vSwitch under both Xen [8] and KVM [11]. Para-virtualized network interfaces were used in all these systems. In the rest of this section, the term "KVM" is used to refer to the performance of Open vSwitch under KVM. Similarly, the term "Xen" is used to refer to the performance of Open vSwitch under Xen's default network I/O architecture. This should not be confused with Hyper-Switch that is also implemented in Xen. The following paragraphs provide more information on the default implementation of Open vSwitch in both these platforms.

Open vSwitch under Xen's Default Network I/O Architecture

In Xen, Open vSwitch is implemented entirely in the driver domain. Here Open vSwitch's data plane is implemented inside the driver domain kernel and its control plane is implemented in user space within the driver domain. Under Xen's default network I/O architecture, all network packets are forwarded to the driver domain, where they are switched. Xen's backend driver called *netback* acts as an intermediary between the guest VMs and the virtual switching module in the driver domain. So netback receives all the packets from the guest VMs and pushes them to Open vSwitch's data plane for switching. Once the switching is complete, netback is once again called upon to deliver the packets to the destination guest VMs.

Netback is multi-threaded, and there is one Linux kernel thread for every vCPU in the driver domain. Each guest VM's virtual network interface (vNIC) is bound to one of these kernel threads. The packets associated with a particular vNIC are processed only by the kernel thread to which it is bound. Further, this binding does not change as long as a vNIC is up and running. Consequently, there is no dynamic load-balancing across the driver domain's vCPUs. This is another downside to Xen's default network I/O architecture.

The recommended practice is to dedicate processor cores for running the driver domain's vCPUs. This means that each vCPU is pinned to a separate processor core by the hypervisor and therefore, the scheduler is forced to run the vCPUs only on those cores. In this evaluation, the driver domain was configured with up to 8 vCPUs.

Open vSwitch under KVM's Vhost-net Architecture

In KVM, Open vSwitch is implemented entirely in the hypervisor (also referred to as the *KVM host*). Here Open vSwitch's data plane is implemented inside the host's kernel and its control plane is implemented in user space within the host. Under KVM's default network I/O architecture, all network packets are forwarded to the host, where the Qemu process running in the host's user space receives them. Then the packets are pushed to Open vSwitch's data plane inside the kernel. But the performance under this architecture was poor, since it involved many user-kernel transitions inside the host. So the *vhost-net* architecture was introduced to improve performance by bypassing the user-level Qemu process.

The vhost-net driver runs inside the host kernel and acts as the intermediary between the guest VMs and the virtual switching module. This is similar to Xen's netback. But unlike netback, there is a separate kernel thread in vhost-net for every vNIC in the system. So this architecture can potentially achieve better load balancing than under Xen's netback. The vhost-net's kernel threads can also be run on dedicated processor cores.

4.5.2 Experimental Results

Inter-VM Performance and Scalability

In these experiments, network performance was studied under different loads by setting up network traffic between VMs co-located on the same server.

Single VM Pair. In the first set of experiments, the network performance was studied by setting up traffic flows between just a single pair of VMs. Further, each guest VM's vCPU was pinned to a separate processor core within the same processor node to avoid any potential VM scheduling effects. Then the Hyper-Switch's performance was compared with the performance of Open vSwitch under both Xen and KVM. Xen's driver domain was configured with 2 vCPUs. Recall that there is one netback kernel thread for every vCPU in Xen's driver domain. Therefore, the packets from a particular VM are always processed by a specific kernel thread in netback on a specific vCPU in the driver domain. In these experiments, the driver domain's vCPUs were also pinned to separate processor cores, but on the same processor node where the corresponding guest VMs' vCPUs were pinned. Similarly, under KVM, the two vhost-net kernel threads (one per guest VM) were also pinned to the same node where the corresponding guest VMs' vCPUs were pinned. Table 4.4 summarizes the experimental setup under all the architectures.

Figures 4.9 and 4.10 show the throughput and latency results from these experi-

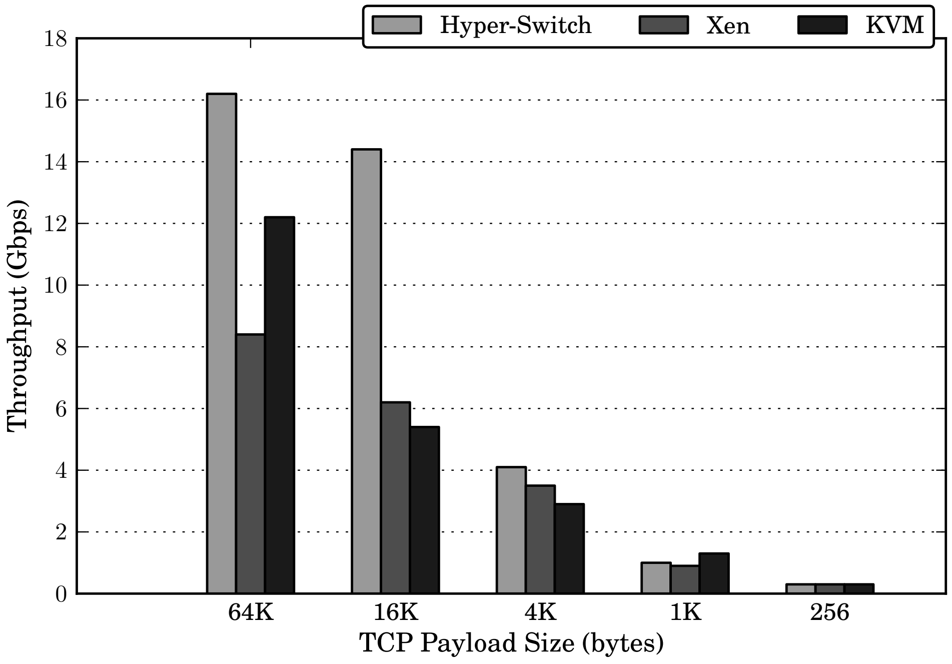


Figure 4.9 : *Inter-VM throughput performance evaluation - between one pair of VMs.*

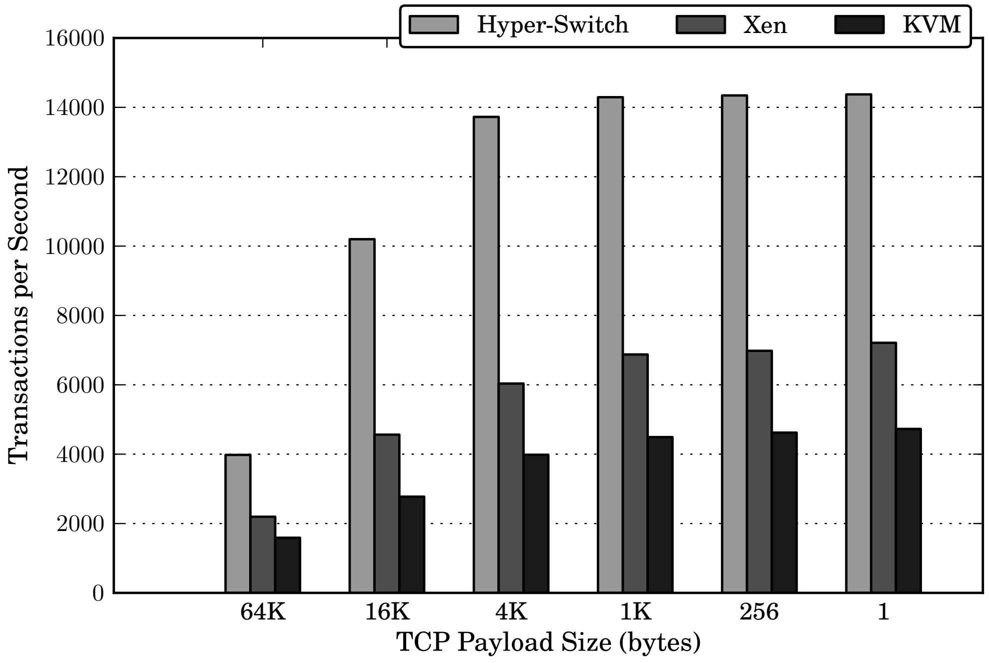


Figure 4.10 : *Inter-VM latency performance evaluation - between one pair of VMs.*

Table 4.4 : *Inter-VM single pair experiments - pinning configuration*

vCPU/Kernel Thread	**Placement on CPU core (logical)**[11]		
	Hyper-Switch	KVM	Xen
Guest 1 vCPU	31	31	31
Guest 2 vCPU	29	29	29
vhost-net kthread/Driver Domain vCPU 1	-	27	27
Vhost-net kthread/Driver Domain vCPU 2	-	25	25

ments. First, as shown in Figure 4.9, higher throughput was achieved under Hyper-Switch than under both the existing architectures in the experiments where the TCP payload was between 4 KB and 64 KB, with stream-based traffic. On average[10], the throughput under Hyper-Switch, in these cases, was ∼56% higher than that under Xen and ∼61% higher than that under KVM. But there was not much performance difference at smaller packet sizes since in those experiments the performance bottleneck was at the sender in all the architectures. Further, contrary to our expectations, the performance under KVM was not consistently higher than the performance under Xen's default network I/O architecture.

Second, as shown in Figure 4.10, higher transactions per second was achieved under Hyper-Switch, across all TCP payload sizes, with request-response TCP traffic. A transaction comprises of a single request followed by a single response in the opposite direction. So these results indicate that the round-trip packet latencies were the lowest under Hyper-Switch among all the three architectures. On average, the

[11]8n...8n + 7, where n = 0...3, form the processor nodes.
2n...2n + 1, where n = 0...15, form the processor modules.

[10]Geometric mean was used to calculate the averages to avoid biasing the average against larger values in the samples.

transactions per second under Hyper-Switch was ~106% higher than that under Xen and ~212% higher than that under KVM. So the Hyper-Switch architecture is suited for both bulk and latency sensitive network traffic. Further, these results show the benefit from optimizations such as preemptive copying and immediate notification of blocked VMs that enable timely delivery of packets.

Pairwise Scalability Experiments. In the next set of experiments, the performance scalability of the three architectures was studied by setting up TCP stream-based traffic flows between 1–16 pairs of VMs in one direction. TCP payload size of 64 KB was used in all the subsequent experiments. Similar to the previous set of experiments, the guest VMs' vCPUs, the vhost-net kernel threads (KVM), and the driver domains' vCPUs (Xen) were pinned to specific processor cores to avoid any VM scheduling effects. Further, the VMs that were communicating with each other were always pinned to the same processor node. Also, the pinning was done such that, in each experiment, the load was uniformly distributed across all the processor modules and nodes in the system. The exact configuration used in each of the three architectures is shown in Table 4.5. The following paragraphs provide an intuition to the rationale behind the pinning configurations.

- **Hyper-Switch:** As each guest VM was added to the system, its vCPU was pinned to a processor core such that all the currently running vCPUs were uniformly distributed across all the modules *and* across all the processor nodes in the system. Therefore, when there were 8 pairs of VMs (1–16) in the system, exactly one of the processor cores in each of the modules was used. When the system was scaled beyond 8 pairs of VMs, the additional guest VMs (16–32) were pinned to the unused core in each of the modules.

Table 4.5 : *Inter-VM scalability experiments - pinning configuration*

vCPU/Kernel Thread	**Placement on CPU core (logical)**[12]		
	Hyper-Switch	KVM	Xen
Guest 1 vCPU	31	31	27
Guest 2 vCPU	29	27	25
Guest 3 vCPU	25	23	19
Guest 4 vCPU	23	19	17
Guest 5 vCPU	15	15	11
Guest 6 vCPU	13	11	9
Guest 7 vCPU	7	7	3
Guest 8 vCPU	5	3	1
Guest 9 vCPU	27	0	26
Guest 10 vCPU	25	4	24
Guest 11 vCPU	19	8	18
Guest 12 vCPU	17	12	16
Guest 13 vCPU	11	16	10
Guest 14 vCPU	9	20	8
Guest 15 vCPU	3	24	2
Guest 16 vCPU	1	28	0
Guest 17 vCPU	30	30	30
Guest 18 vCPU	28	31	28
Guest 19 vCPU	26	27	22
Guest 20 vCPU	24	23	20
Guest 21 vCPU	22	15	14
Guest 22 vCPU	20	13	12
Guest 23 vCPU	18	7	6
Guest 24 vCPU	16	5	4
Guest 25 vCPU	14	27	27

Continued on next page

Table 4.5 : *Inter-VM scalability experiments - pinning configuration (continued)*

vCPU/Kernel Thread	**Placement on CPU core (logical)**[12]		
	Hyper-Switch	KVM	Xen
Guest 26 vCPU	12	25	25
Guest 27 vCPU	10	19	19
Guest 28 vCPU	8	17	17
Guest 29 vCPU	6	11	11
Guest 30 vCPU	4	9	9
Guest 31 vCPU	2	3	3
Guest 32 vCPU	0	1	1
Vhost-net kthread/Driver Domain vCPU 1	-	29	29
Vhost-net kthread/Driver Domain vCPU 2	-	25	25
Vhost-net kthread/Driver Domain vCPU 3	-	21	21
Vhost-net kthread/Driver Domain vCPU 4	-	17	17
Vhost-net kthread/Driver Domain vCPU 5	-	13	13
Vhost-net kthread/Driver Domain vCPU 6	-	9	9
Vhost-net kthread/Driver Domain vCPU 7	-	5	5
Vhost-net kthread/Driver Domain vCPU 8	-	1	1
Vhost-net kthread 9	-	2	-
Vhost-net kthread 10	-	6	-
Vhost-net kthread 11	-	10	-
Vhost-net kthread 12	-	14	-
Vhost-net kthread 13	-	18	-
Vhost-net kthread 14	-	22	-
Vhost-net kthread 15	-	26	-
Vhost-net kthread 16	-	30	-
Vhost-net kthread 17	-	29	-
Vhost-net kthread 18	-	25	-

Continued on next page

Table 4.5 : *Inter-VM scalability experiments - pinning configuration (continued)*

vCPU/Kernel Thread	**Placement on CPU core (logical)**[12]		
	Hyper-Switch	KVM	Xen
Vhost-net kthread 19	-	21	-
Vhost-net kthread 20	-	17	-
Vhost-net kthread 21	-	13	-
Vhost-net kthread 22	-	9	-
Vhost-net kthread 23	-	5	-
Vhost-net kthread 24	-	1	-
Vhost-net kthread 25	-	2	-
Vhost-net kthread 26	-	6	-
Vhost-net kthread 27	-	10	-
Vhost-net kthread 28	-	14	-
Vhost-net kthread 29	-	18	-
Vhost-net kthread 30	-	22	-
Vhost-net kthread 31	-	26	-
Vhost-net kthread 32	-	30	-

[12] 8n...8n + 7, where n = 0...3, form the processor nodes.

2n...2n + 1, where n = 0...15, form the processor modules.

- **KVM:** Here, the pinning was again performed using a similar approach as in the first case to distribute the load evenly across all the modules and nodes in the system. But recall that, under KVM, each guest VM is associated with a vhost-net kernel thread in the host. Further, both the guest VM's vCPU and the vhost-net thread were pinned to separate processor cores. Therefore, in this case, all the processor cores were used up when there were just 16 guest VMs in the system (unlike the first case).

  So when there were 4 pairs of VMs (1–8), one of the cores in all the processor modules was busy. When additional VMs (9–16) were added to the system, their vCPUs and vhost-net threads were pinned to the unused core in each of the modules. When there were 8 pairs of VMs, *all* the processor cores in the system were used up. Therefore, when the system was scaled up beyond 8 pairs of VMs, each processor core had to run one of the guest VMs' vCPU *and* one of the vhost-net threads.

- **Xen:** Xen's driver domain was configured with 8 vCPUs. These vCPUs were pinned to dedicated processor cores in accordance to the best practices on Xen deployment. The driver domain's vCPUs were distributed across all the nodes by pinning two of them to each processor node in the system. Then the guest VMs' vCPUs were evenly distributed across the remaining processor cores using an approach that was similar to the first two cases. But unlike the second case, only some of the processor cores had to run more than one guest VM's vCPU.

The results in Figure 4.11 show that the Hyper-Switch architecture exhibited much better performance scalability than both the existing architectures. Specifically, under Hyper-Switch, the performance reached a peak throughput of ∼81 Gbps before

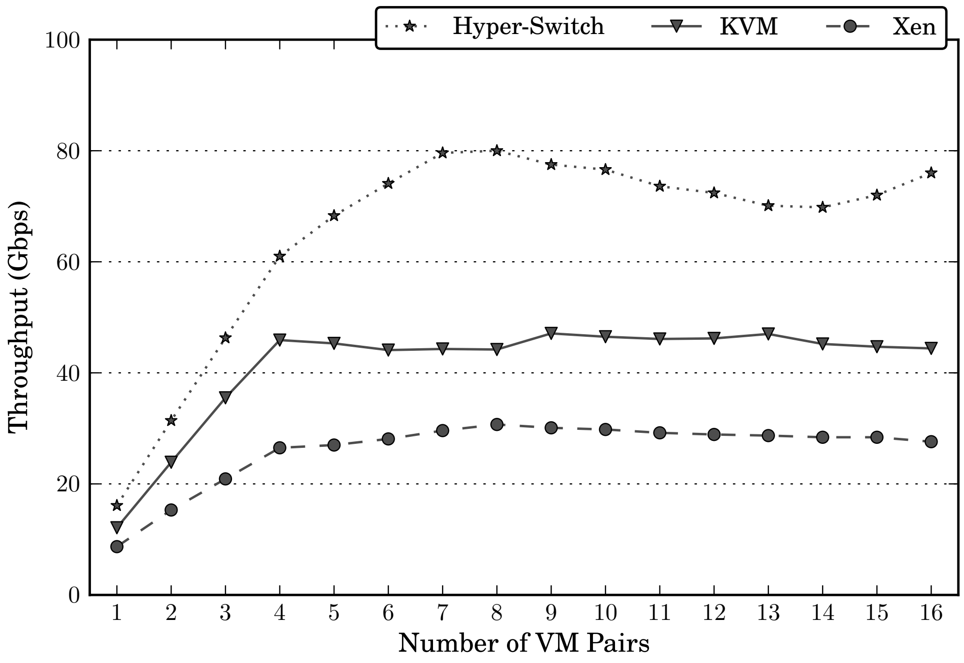


Figure 4.11 : *Pairwise performance scalability evaluation - throughput.*

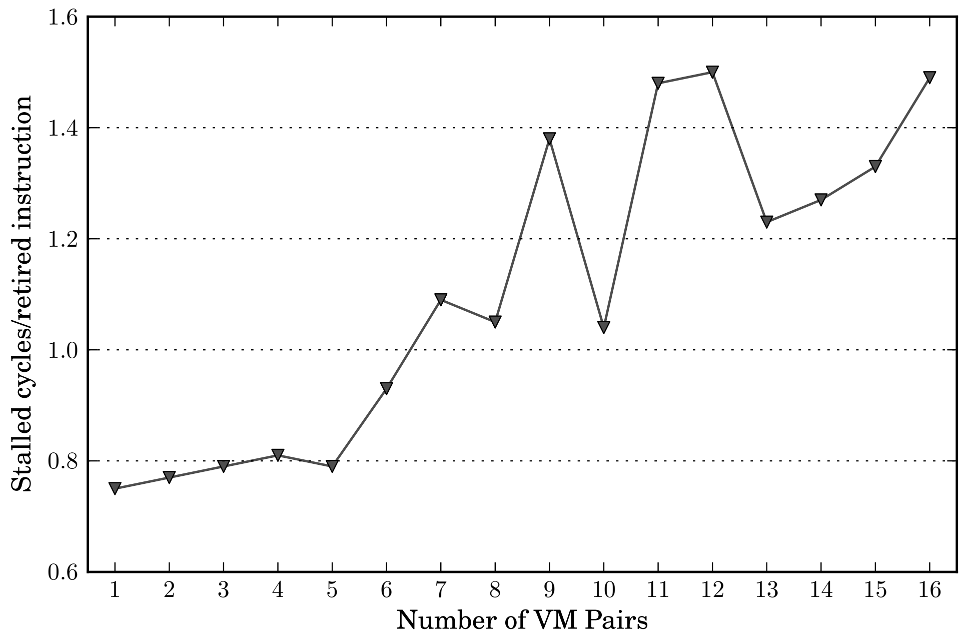


Figure 4.12 : *Pairwise performance scalability evaluation - processor stalls during reads and writes.*

it started to flatten out. But the peak throughput was only ∼47 Gbps and ∼31 Gbps under KVM and Xen respectively. Further, the performance under these existing architectures did not scale beyond 4 pairs of VMs. Though it should be noted that, while their performances did not scale well, it remained relatively stable after reaching the peak throughputs. On average, the throughput under Hyper-Switch was ∼55% higher than that under KVM and ∼146% higher than that under Xen.

The performance curve under Hyper-Switch was further studied to understand and identify the performance limits. There were three distinct regions in Hyper-Switch's performance curve as shown in Figure 4.11:

- In the first region between 1 and 4 pairs of VMs, the performance scaled almost linearly, from ∼16.2 Gbps to ∼62.7 Gbps.
- In the second region between 5 and 7 pairs of VM, the performance continued to scale linearly but at a lower rate (reduced slope), from ∼62.7 Gbps to ∼81 Gbps.
- In the third region, beyond 8 pairs of VMs, the performance did not scale further.

Fundamentally, the network performance is determined by the number of packets that can be transferred between the source and destination VMs in a given time. A typical packet transfer involves switching and packet copying overheads. Further, the packet copying overheads might be incurred multiple times. Fundamentally, the packet has to be copied from the source VM to the destination VM. Generally, there is another packet copy inside the destination VM when the packet is transferred between the kernel and application buffers[13]. But there are limits to how many packets

[13]A similar packet copy can also occur in the transmit VM. But this was not the case in these experiments since the `sendfile` technique was used to transmit packets.

that can be processed by a single processor. This is determined in part by the underlying hardware (processor) architecture. The hardware architecture determines how efficiently the available processor time is used to process—switch and copy—packets. Today's processors are incredibly complex and therefore, there are several factors that impact this efficiency. In particular, the structure of the memory subsystem can have a significant impact on performance [57]. This includes the size and levels of the CPU caches, the maximum number of outstanding reads/writes/cache misses, the available memory bandwidth, the number of channels to the system memory, and so on.

One can scale the performance beyond the limits imposed by a single processor core by increasing concurrency, *i.e.,* by using multiple processor cores. But some of the system resources could be shared between processor cores—such as CPU caches, memory channels, etc.—that could potentially reduce the available concurrency. When additional VMs are added to the system, there is a natural increase in concurrency since many of the switching tasks can be concurrently performed under each VM's context. Further, under Hyper-Switch, the offloading of packet processing adds to this concurrency. The Hyper-Switch's *offload criteria* determines when and where the packet processing is offloaded. The criteria are explained in detail later in this section. In short, when choosing an idle core for offloading packet processing, preference is given to idle processor cores on the same node as the receiving VM's vCPU. This is done to take advantage of any CPU cache locality. Recall that the cores within a node share the last level cache (see Figure 4.8). Further, the packet processing is offloaded only to a processor core in an idle module, *i.e.,* a module where both the processor cores are idle. This is done to avoid potential cache interference effects. Recall that the cores within a module share the instruction caches and the second level data cache (see Figure 4.8).

In the first region of the curve (in Figure 4.11), each pair of VMs were run in a separate processor node. Further, the packet processing was offloaded to other cores within the same node according to the offload criteria. So under these conditions, the best scalability was achieved. In the second region, some of the packet processing had to be offloaded to idle modules on other nodes in the system. This was not as efficient since packets had to be copied across processor nodes. Hence the performance scalability was reduced. In the third region, the performance stopped scaling in part due to the reduction in the offloading of packet processing since most of the processor modules were busy. Also, some of the VMs' vCPUs were running on two cores within the same processor module. So the cache interference effects also came into effect. Finally, as more VMs were added to the system, there was increased contention for the system resources such as the CPU caches. So, effectively, all these factors offset the increase in packet processing concurrency and as a result, the performance flattened out.

Unfortunately, it was not possible to determine the one factor that limited performance. Oprofile [46] was used to measure several hardware events such as cache misses at various levels, system reads and writes, etc. It is highly likely that a combination of factors related to the memory subsystem ultimately determined the performance curve. Previous work has shown that the structure of the underlying memory subsystem can have a huge impact on the system performance [57]. Figure 4.12 plots the number of CPU cycles that reads and writes were stalled per retired instruction as the number of VM pairs were increased. Clearly, there was increased contention for system resources that led to more processor stalls across the system. While these results do not conclusively prove the hypothesis, they indicate that the memory subsystem had a critical role in determining the performance.

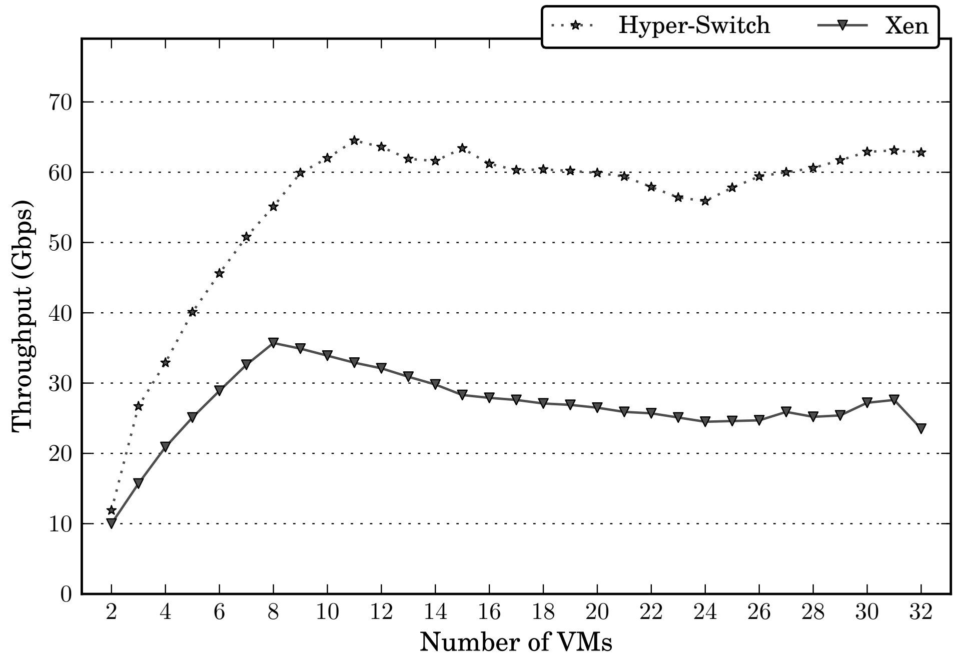


Figure 4.13 : *All-to-all performance scalability evaluation - throughput.*

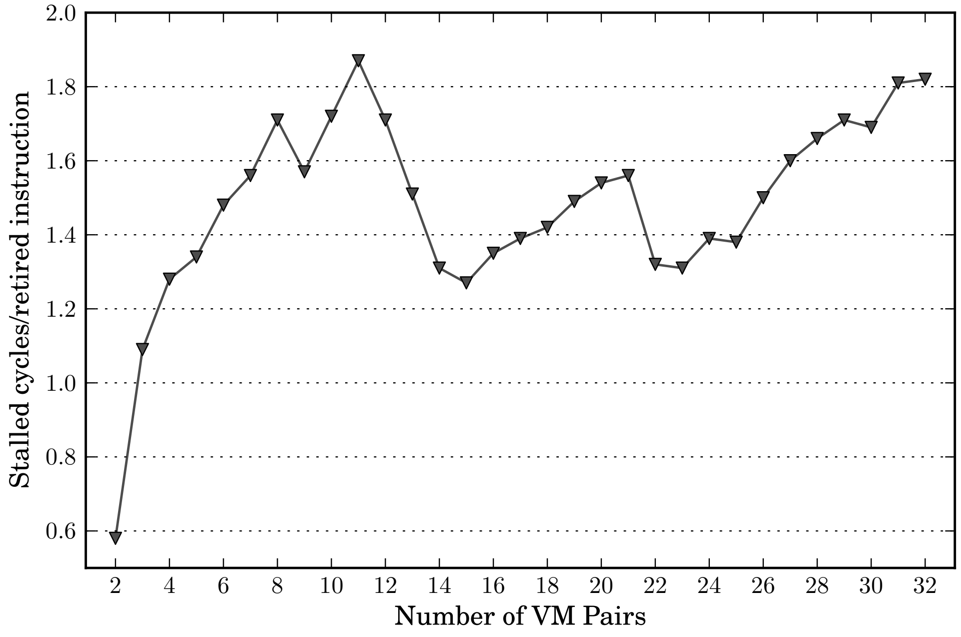


Figure 4.14 : *All-to-all performance scalability evaluation - processor stalls during reads and writes.*

All-to-all Scalability Experiments. In the second set of scalability experiments, TCP stream-based network traffic was setup between every pair of VMs in the system in both the directions. These experiments were designed to generate significant load on the network by having tens of VMs concurrently communicating with each other. For instance, when there were 30 VMs in the system, there were as many as 870 concurrent TCP flows between all the VMs. Again, the Hyper-Switch's performance was compared with the performance under both KVM and Xen. The configuration and setup was similar to the previous set of experiments where all the guest VMs' vCPUs, the driver domains' vCPUs (Xen), and the vhost-net kernel threads (KVM) were pinned to one of the processor cores in the system (as described in Table 4.5).

Figure 4.13 shows the results from these scalability experiments. The performance again scaled much better under Hyper-Switch than under KVM or Xen. Specifically, under Hyper-Switch, the performance reached a peak throughput of ~65 Gbps as compared to ~55 Gbps and ~31 Gbps under KVM and Xen respectively. Further increasing the number of VMs did not degrade performance. Instead, the performance remained relatively stable.

Similar to the previous set of experiments, the performance curve under Hyper-Switch scaled up very well at the beginning before tapering off. The performance analysis presented with the previous results is applicable here as well. In fact, the contention for system resources is even higher in this case, due to the significant load placed on the system. Hence the performance scaled only till 11 VMs where the peak throughput was achieved. Figure 4.14 plots the number of CPU cycles that reads and writes were stalled per retired instruction as the number of VM pairs were increased in these experiments. Processor stalls were again observed, that indicated increased contention for system resources.

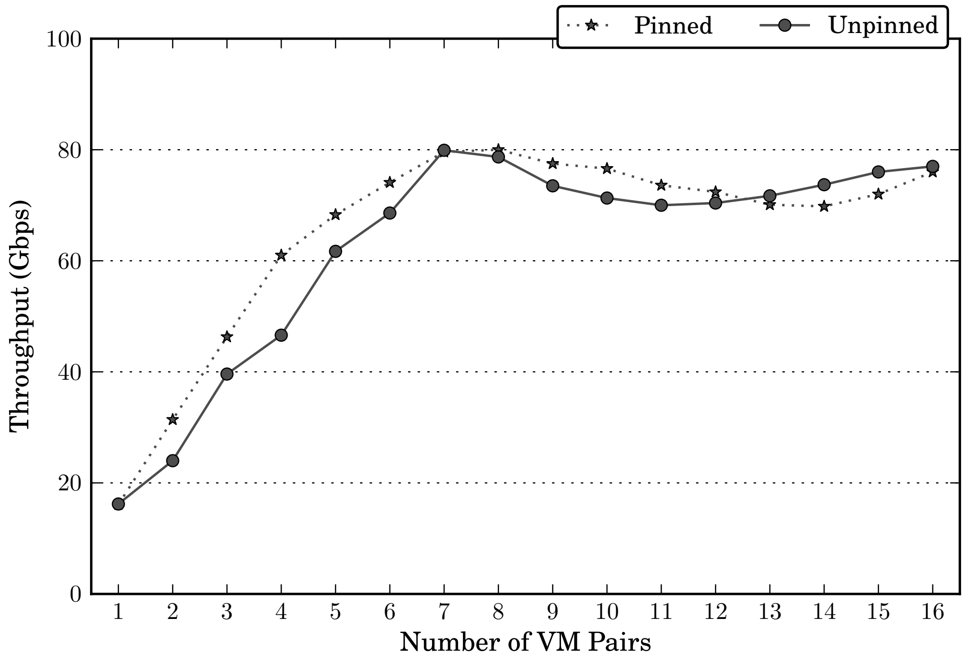


Figure 4.15 : *Effect of pinning under Hyper-Switch - pairwise experiments.*

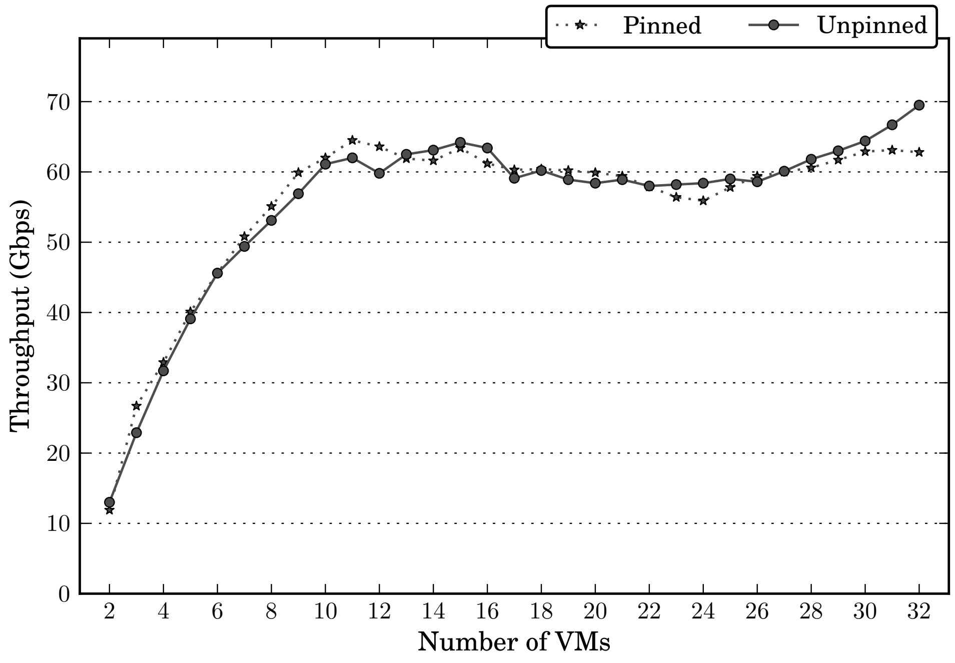


Figure 4.16 : *Effect of pinning under Hyper-Switch - all-to-all experiments.*

Effect of Pinning. In all the experiments described thus far, the guest VMs' vCPUs were pinned to one of the processor cores. This was primarily done to avoid any VM scheduling effects so that the results were comparable. In the next set of experiments, the impact of pinning on the performance under Hyper-Switch was studied. This was done by repeating the scalability experiments without pinning and then comparing the results with those obtained with pinning.

Figures 4.15 and 4.16 show the results from both the pairwise and the all-to-all experiments. The effect of pinning was more pronounced in the pairwise experiments especially in the first half of the performance curve where the performance was higher when the guest VMs' vCPUs were pinned. Specifically, the average percentage difference in throughput between the pinned and unpinned configurations was ∼20% when the number of VM pairs were between 3 and 7. Interestingly, almost the same peak throughput was seen under both the configurations when there were 8 VMs in the system. However, in the all-to-all scalability experiments, the performance curves under both the configurations were very similar with an average percentage difference in throughput just around 3%.

The reason for reduced performance under the unpinned configuration, especially in the pairwise experiments, was due to the way the Xen scheduler allocated processor cores to run the guest VM's vCPUs. Since the guest VM's vCPUs were not pinned, it was up to the Xen scheduler to allocate any available processor cores. It was observed that the scheduler allocated one core from each module, in order, across all the nodes before wrapping around and allocating the other core in each of the modules. For instance, when there were 4 guest VMs in the system, one core from each of the 4 modules in the first processor node were allocated. As a result, the packet processing was often offloaded to other (remote) processor nodes in the system. But under

Table 4.6 : *External latency performance evaluation.*

	Hyper-Switch	Xen	KVM
Transactions per second	13,243	11,342	10,721

the pinned configuration, the processor cores from each node were allocated in an interleaved manner. As a result, the packet processing could be offloaded to an idle core on the same node most of the time (until the system was scaled beyond 8 pairs of VMs). Therefore, the offloading of packet processing was more efficient under the pinned configuration.

In the all-to-all experiments, packets were copied across processor nodes even under the pinned configuration since all the VMs were communicating with one another. So the processor core allocation to the guest VMs' vCPUs did not affect performance. In summary, under low load, processor configuration had an impact on performance. But the impact was significantly reduced under higher loads.

External Performance

In the external experiments, the network traffic was setup between guest VM(s) and the external client. The Hyper-Switch's performance was again compared with the performance under KVM and Xen. The driver domain, under Hyper-Switch and Xen, was configured with only a single vCPU. But under KVM, the vhost-net kernel threads in the host, could be run on any of the available processor cores. In the

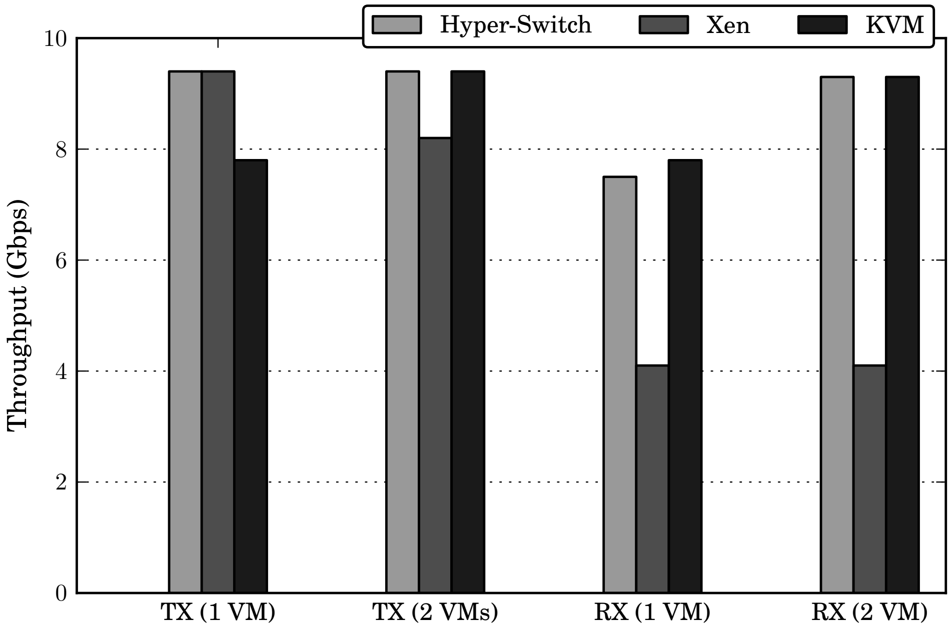


Figure 4.17 : *External throughput performance evaluation.*

TX experiments[14], there were one or two guest VMs (concurrently) sending packets to the external client. In the RX experiments, there were one or two guest VMs (concurrently) receiving network packets from the external client. The guest VMs' vCPUs and the driver domain's vCPU were pinned to separate processor cores on the same processor node.

The results in Figure 4.17 show that the Hyper-Switch's performance was comparable and in some cases even better than the performance under both KVM and

[14]The `sendfile` technique was not used in these experiments since anomalous results were observed under Hyper-Switch and Xen. Specifically, the TCP segments that were being transmitted were, for unknown reasons, much smaller than the expected 64 KB. Consequently, the overheads incurred at the sender were significantly higher.

Xen. In the TX experiments, with a single guest VM transmitting packets, line rate of ∼9.4 Gbps was achieved under both Hyper-Switch and Xen. But under KVM, the TX VM's vCPU was a performance bottleneck. Therefore, only ∼7.8 Gbps was possible in this case. But with two guest VMs on the transmit side, the scenario was reversed. Under KVM, line rate was achieved. But under Xen, the throughput dropped to ∼8.2 Gbps. Recall that the driver domain in Xen was configured only with a single vCPU. Therefore, there was only one kernel thread in netback to process packets. So the single netback thread had to process packets from more than one traffic flow. This increased the per-packet overhead since optimizations like batching, which reduce the per-packet overhead, were not as effective. This effect was not observed under KVM, since the per-VM backend vhost-net threads could be run on more than one processor core in the host.

In the RX experiments, with one guest VM receiving packets, the CPU at the guest VM was the bottleneck. So line rate was not achieved under any of the architectures. But the performance was better under Hyper-Switch and KVM. But with two guest VMs receiving packets, line rate of ∼9.4 Gbps was achieved under Hyper-Switch and KVM. Under Xen, the driver domain's vCPU was the performance bottleneck. Therefore, having a second guest VM receive packets had no positive impact on the aggregate throughput. But at the same time, unlike the TX experiments, it did not reduce the throughput as well. These results show that the driver domain under Hyper-Switch can send and receive packets at 10 GbE line rate using a single CPU core. So the driver domain consumes minimal resources.

Table 4.6 shows the latency results from the experiments where request-response TCP traffic was setup between a single guest VM and the external client. These results show that higher transactions per second was achieved under Hyper-Switch.

As explained before, higher transactions per second indicate lower round trip latency. Therefore, despite the “longer” route taken by packets under Hyper-Switch due to their forwarding through both the hypervisor and the driver domain, the packet latencies were still the lowest under Hyper-Switch. These results also confirm the benefits from the various optimizations described in Section 4.3.

Design Evaluation

In the next set of experiments, the packet processing offload criteria were evaluated. In these experiments, network traffic was setup between a single pair of VMs in one direction. The VMs’ vCPUs were pinned to separate processor cores. Further, the packet processing was offloaded to a single dedicated processing core to enable proper evaluation. First, the effect of pinning the TX and RX guest VMs’[15] vCPUs to the same node that contained the dedicated processing core was studied. The following configurations were considered:

- `No offload`: Packet processing was not offloaded.
- `TX and RX`: Both the vCPUs were pinned to the same node that contained the dedicated processing core.
- `RX only`: Only the RX VM’s vCPU was pinned to the same node that also contained the dedicated processing core.
- `TX only`: Only the TX VM’s vCPU was pinned to the same node that also contained the dedicated processing core.

[15]The so called RX guest VM does more than just receive packets. It has to send back ACKs to the TX guest VM since TCP is used in all the experiments. This fact was ignored for the sake of simplicity and ease of explanation.

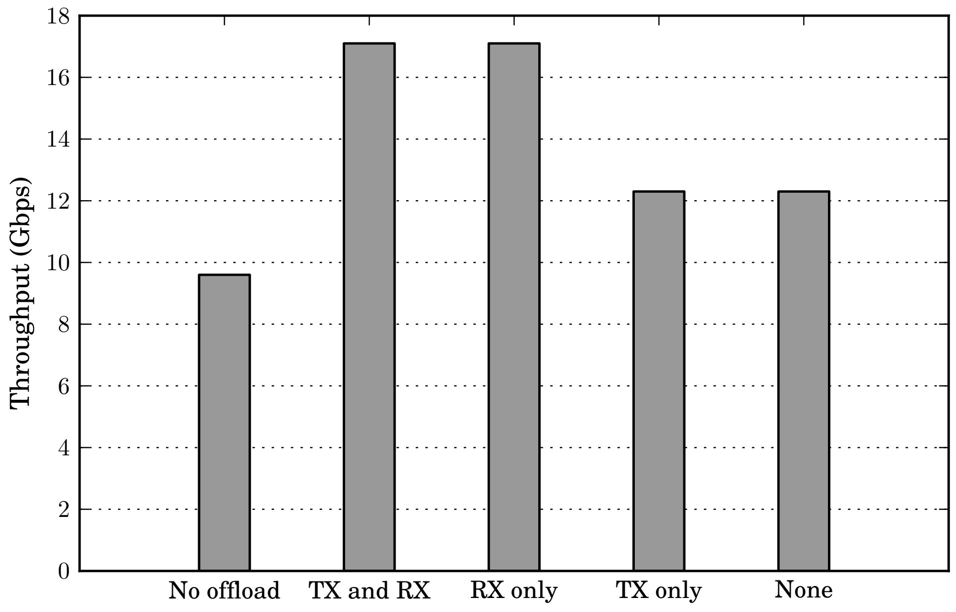


Figure 4.18 : *Offload evaluation I - Impact of offloading packet processing to a core on the same processor node where the TX and/or RX guest VMs were running.*

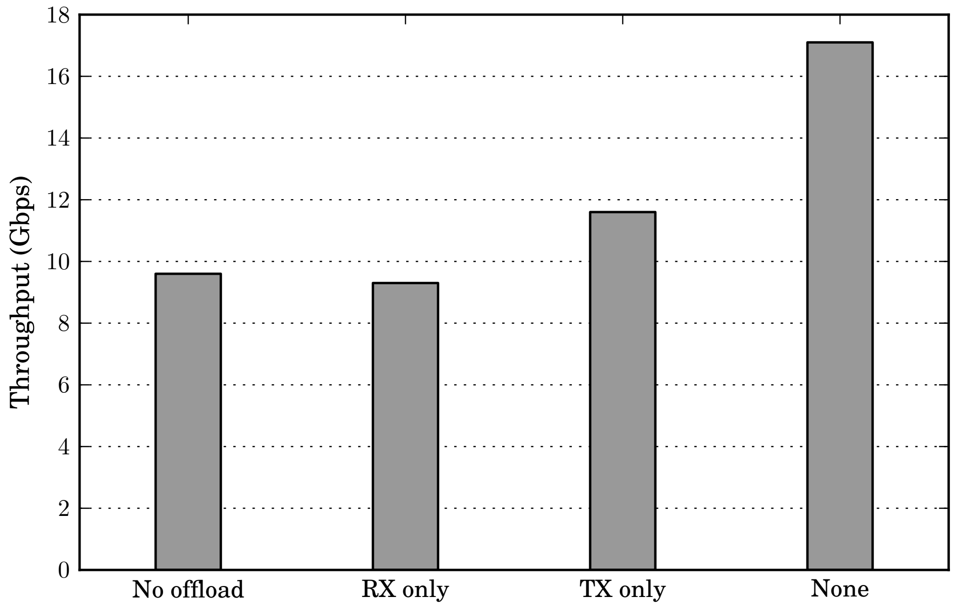


Figure 4.19 : *Offload evaluation II - Impact of offloading packet processing to a core on the same module where TX or RX guest VMs were running.*

- `None`: Neither were pinned to the same node as the dedicated processing core.

In each case, the throughput between a pair of VMs was measured. The results in Figure 4.18 show that the best performance was achieved when the receiving VM was running on the same node that also contained the idle processing core (bars '`TX and RX`' and '`RX`'). They also show that there was a small benefit from offloading to an idle core on another node (bars '`TX only`' and '`None`') than not offloading at all (bar '`No offload`'). So despite the extra costs incurred in moving the packets between the last level caches on different processor nodes, offloading was still beneficial.

Second, the effect of pinning the RX and TX VMs' vCPUs to the same module that contained the dedicated processing core was studied. The following configurations were considered:

- `No offload`: Packet processing was not offloaded.
- `RX only`: Only the RX VM's vCPU was pinned to a processor core on the same module that also contained the dedicated processing core.
- `TX only`: Only the TX VM's vCPU was pinned to a processor core on the same module that also contained the dedicated processing core.
- `None`: Neither of the vCPUs were pinned to a processor core on the same module that also contained the dedicated processing core.

The results in Figure 4.19 show that this can have a negative effect on performance, especially on a receiving VM (bar '`RX`'). In this case, the performance was worse than the performance when the packet processing was not offloaded (bar '`No offload`'). This was due to cache interference effects since the cores within a module shared the instruction caches and the second level data cache. The cache miss data, which was obtained using Oprofile [46], indicated that the shared instruction cache was the primary cause of the interference effects. Such cache interference issues have been previously observed on this processor architecture [58].

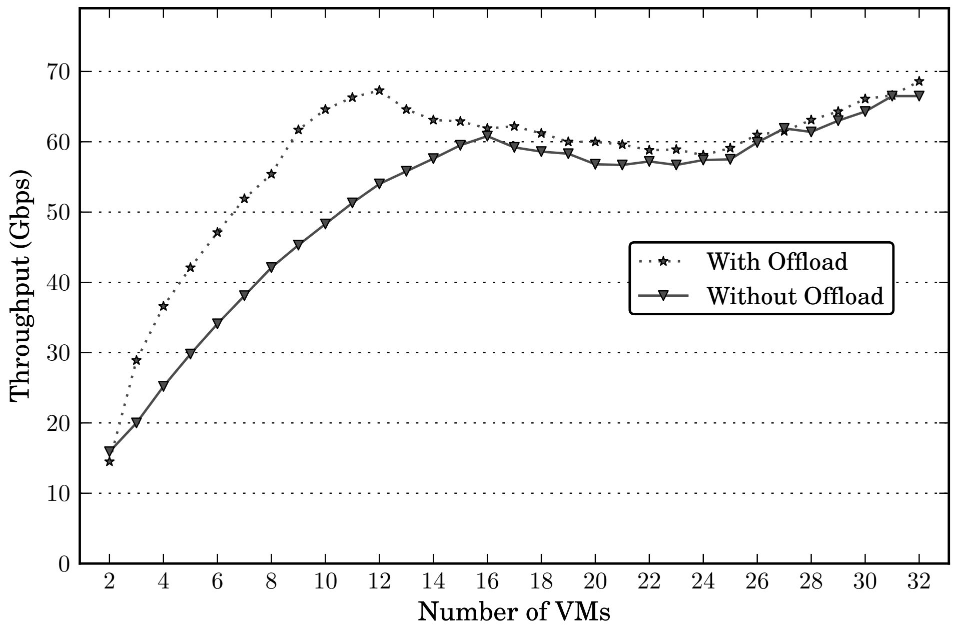


Figure 4.20 : *Hyper-Switch offload evaluation III. Impact of offloading packet processing on performance scalability.*

In summary, based on these results, the offload criteria were designed as follows:

- First, the processor node on which the receiving VM's vCPU was running was searched for an idle module. If an idle module was available, then one of its processor cores was chosen.

- Else, all the other nodes in the system were searched for an idle module. If an idle module was available, then one of its processor cores was chosen.

- If still a suitable idle core was not available, the packet processing was not offloaded.

The offload criteria could vary depending on a processor's cache hierarchy. So the exact offload criteria should depend on the particular processor that is used to run the Hyper-Switch.

The maximum performance between a single pair of VMs (both pinned to the same processor node) using a dedicated packet processing core was ∼17.2 Gbps. The actual Hyper-Switch performance, *i.e.,* with dynamic offloading, between a pair of VMs was ∼16.2 Gbps (as shown in Figure 4.11). So the mechanism that was used to dynamically offload packet processing was able to achieve 94% of the performance that was obtained using a dedicated packet processing core. These results show that the Hyper-Switch architecture was capable of achieving high performance without the need for dedicated resources.

In the final set of experiments, the impact of offloading packet processing on performance scalability was studied. Here network traffic was setup between every pair of VMs in the system in both the directions. Then the performance scalability was compared with and without the offloading of packet processing. Figure 4.20 shows that the performance with offloading was much higher up to 16 VMs. But beyond that the performance with and without offloading was similar. This was because, when the system was scaled beyond 16 VMs, there was significant load on the system and therefore, the Hyper-Switch could no longer find idle modules to offload packet processing. These results show that the offloading of packet processing leads to better performance, especially when the system is not under heavy load.

4.6 Discussion and Future Work

In this section, ideas and directions for future work are discussed. This includes: (1) offloading of packet copies to DMA engines, (2) using zero-copy transmission for external traffic, (3) better handling of network congestion, and (4) reducing the impact on scheduling latency.

4.6.1 Offloading Packet Copies to DMA Engines

The Hyper-Switch architecture offloads packet copies to idle cores to enable better utilization of the available processor resources and to increase concurrency, as described in section 4.3. Modern server platforms provide an alternate way to offload packet copies using *DMA Engines*. The DMA engines, typically built into the server chipset, perform asynchronous memory-to-memory movement of data. So the Hyper-Switch will take advantage of the DMA engines to perform packet copies from the source to the destination VM. Section 5.2 provides more information on DMA engines.

The packet copies are setup at a DMA engine by creating DMA descriptors in the host memory. The descriptors indicate the source address, the destination address, and the length of the data to be copied. The use of DMA engines will be integrated into the Hyper-Switch architecture as follows. Once a packet is placed in the internal receive queue at a virtual network interface, the DMA engine will be programmed to copy the packet into the receiving VM's memory. This will happen in place of offloading the packet copy to an idle core. Then the receiving VM will check if the packet copy has completed. But the descriptors in the receive ring will have to be accessed when the DMA engine is being programmed. Previously, only the packet copier had to access the receive descriptors. So proper synchronization will be needed when the descriptors are being read.

In general, it has been shown that utilizing such DMA engines is profitable only for large copies since the setup and completion overheads outweigh the benefits [59]. The completion costs arise largely from polling or interrupt overheads. So the Hyper-Switch architecture will use the DMA engine to copy packets only if the packet sizes are more than a particular threshold, say 128 bytes. Further, the completion overheads will be mitigated by polling only when the receiving VM is notified at the end of a receive timer period.

Small packet copies that are not offloaded to the DMA engine will be either performed by the receiving VM or offloaded to idle cores as before. But a potential complication with this scheme is that the packet copies may complete out-of-order. Therefore, similar to the transmit side, two rings will be needed in the receive side as well. Then the receiving VM will queue empty buffers in one of the rings and the Hyper-Switch will queue buffers with the packet data, after they have been copied, in the other ring.

4.6.2 Zero-copy Transmission for External Traffic

In the proposed architecture, the zero-copy technique is used only for packet transmission to the Hyper-Switch. Packets that have to be forwarded to the driver domain for external transmission must be copied into the driver domain's memory. If the zero-copy technique is extended all the way to the device then the packet copy overhead will be entirely eliminated during external packet transmission.

However, if zero-copy is used for external packet transmission, then the driver domain will need explicit access to the packets that are in the guest VM's memory. Xen, for instance, provides the grant mechanism that is used to share memory between

VMs. So the grant mechanism is used by the driver domain to access a guest VM's memory. But the grant mechanism is expensive since it involves costly hypercalls on every packet transmission as explained in Section 3.2. So, in essence, the copying overheads are replaced by the memory sharing overheads, which can potentially make the performance worse.

Instead, ideas from an earlier work [49, 60] (described in Section 3.4) will be adapted to implement zero-copy without incurring the memory sharing overheads. Essentially, the memory sharing will be setup directly by the hypervisor without driver domain participation. The transmitting guest VM will implicitly agree to share the memory pages that contain the packet data when it queues the packet for transmission. This is unlike the original grant mechanism that requires explicit permission by the source VM to share one of its pages.

Then the hypervisor will map the guest pages into the driver domain's address space, thereby providing the driver domain access to the guest VM's memory that contain the packet data. But, additionally, the Hyper-Switch will also be notified once the external packet transmission is completed. Only then will the Hyper-Switch notify the transmitting VM. Otherwise, the transmitting VM might release the pages that contain the packet data, prematurely, before the external packet transmission has been completed.

4.6.3 Handling Congestion

In Hyper-Switch, congestion is inferred at a virtual network interface when its receive ring is full. This will happen when a receiving VM is too slow in processing packets. But if the transmitting VMs keep sending packets, it will lead to a large number of packets pending in the internal receive queue at the congested virtual network interface. This is not acceptable since the pending network packets will potentially take up all the available memory in the hypervisor.

A potential solution is to cap the number of pending packets at each virtual network interface. Packets are allocated for transmission only if the packet cap has not been reached. If the cap has been reached, then the descriptors in the transmit ring will not be consumed. Eventually, the transmitting VM will stop queuing packets.

But this solution will not suffice if the congestion is artificially created by one or more malicious VMs. Let's say one or more VMs simply refuse to receive packets. Then the packet cap will be reached at all the VMs that are sending packets to the malicious VMs. As a result, the VMs will not be able to transmit any more packets. Therefore, the solution is to drop packets, when the packet cap is reached. Further, the packets that are allocated the earliest will be dropped first since they are most likely to be stuck in one of the internal receive queues.

Additionally, as an optimization, a virtual network interface will be *masked* when congestion is detected. The idle cores will not process packets from an interface that is masked. So masking avoids the scenarios when packet copies are attempted at a virtual network interface only to find that its receive ring is full due to congestion or packet transmissions are attempted only to find that the packet cap has been reached.

4.6.4 Reducing the Impact on Scheduling Latency

A potential downside from using idle cores to perform packet processing is that it might have a negative impact on the scheduling and running of guest VMs. In Xen, for instance, the scheduler is run in a softirq context. Now let's say a particular processor core is idle and therefore, halted. When a VM has to be scheduled to run on that core, the scheduler softirq is marked pending and the core is signaled via an

IPI. When the core wakes up, it handles all the pending softirqs from the idle loop. As a result, the scheduler is executed and the VM is run on that core.

Recall that the Hyper-Switch uses the idle loop to also perform any offloaded packet switching or copying. Consequently, the idle loop cannot handle the pending softirqs while the packets are being processed. Therefore, the scheduler will not be run immediately. When a core is signaled using an IPI, it merely wakes up the core and does not force the scheduler to be run. So if the softirqs are not handled immediately, the scheduling and running of VMs will be delayed.

We partially mitigate this problem by checking for pending softirqs after each packet is processed. This means that the scheduler will be delayed by at most the time taken to process—switch or copy—a single packet. But packet processing latency will vary depending on several factors like the size of the packet, whether the packet data is in the cache or not, and so on. Therefore, the resulting unpredictable delays might be unacceptable.

A potential solution to this problem is to terminate the packet processing as soon as a processor core is notified that the scheduler has to be run. But aborting packet switching or copying in the middle of an operation might not be always feasible since it might leave the system in an inconsistent state. Instead, the packet processing will be performed as a transaction by dividing the operations into two phases. In the first phase, the bulk of the switching or copying operations will be performed. In the second phase, the operations will be committed. During packet switching, this will entail consuming the descriptors on the transmit ring and placing the switched packet in the internal receive queue at the destination virtual network interface. During packet copying, this will entail removing the packet from the internal receive queue and consuming the descriptors on the receive ring. The commit phase will be performed atomically.

4.7 Conclusions

Today, the most widely used network I/O virtualization architectures are the ones where the I/O devices are virtualized in software. But these existing architectures suffer from several limitations. The hypervisor-based architectures are susceptible to serious faults due to the inflated size of their trusted computing base. The driver domain architectures incur significant software overheads that not only reduce the achievable I/O performance but also severely limit I/O scalability. While the new memory sharing mechanism presented in the previous chapter reduces these overheads to a certain extent, there are fundamental issues with traditional driver domain architectures. For instance, the driver domain has to be scheduled in a timely manner to avoid unpredictable delays in packet processing. Further, it requires CPU resources to be dedicated to ensure good performance. Moreover, proper CPU usage accounting using driver domains is hard.

This chapter presented the Hyper-Switch architecture that combines the best of the existing architectures. It hosts the device drivers in a driver domain to isolate any faults and the last hop virtual switch in the hypervisor to perform efficient packet switching. In particular, the hypervisor implements just a fast, efficient data plane of a flow-based software switch. Further, the driver domain is needed only for handling external network traffic.

This chapter also described several optimizations that enable high performance. The VM state-aware batching of packets mitigated the cost of hypervisor entries and guest notifications. The preemptive copying and immediate notification of blocked guest VMs reduced packet processing latency. Dynamic offloading of packet processing to idle CPU cores in the system increased concurrency and made better use of the available CPU resources.

Finally, this chapter also presented a detailed evaluation of the Hyper-Switch architecture using a prototype implemented in the Xen platform. The evaluation showed that the Hyper-Switch outperformed both Xen's default network I/O architecture and KVM's vhost-net architecture. Hyper-Switch's performance was superior both in terms of the absolute bandwidth and the scalability as the number of VMs and traffic flows were varied. For instance, in the pairwise scalability experiments the Hyper-Switch achieved a peak net throughput of $\sim$81 Gbps as compared to only $\sim$31 Gbps and $\sim$47 Gbps under Xen and KVM respectively. The Hyper-Switch architecture has shown that it is possible to support high performance virtual switching entirely in software.

Chapter 5

sNICh : Leveraging Switch-Server Integration

5.1 Introduction and Motivation

The existing solutions for last hop virtual switching in datacenters that deploy machine virtualization represent two ends of the spectrum: switching entirely in software within the server and switching in hardware outside the server. These solutions are inefficient as they either incur significant software overheads or waste network link bandwidth. The Hyper-Switch architecture, described in Chapter 4, reduces some of the overheads associated with software switching. For instance, it eliminates the memory sharing overheads that are typical in I/O architectures that support driver domains. However, there are overheads inherent in software switching that behooves the exploration of other points in the last hop virtual switching design space.

A middle ground in this space is to switch packets within the server's network interface cards (NICs). By moving the network switching functionality from the software to the network interface hardware, the software overheads of switching have been largely eliminated. Further, this solution also avoids any wastage of external link bandwidth since the packets are switched entirely within the server. However, the NICs that implement this solution only support rudimentary switching functionalities.

We propose a new and efficient solution for last hop virtual switching called the *sNICh* architecture. As the name implies, the sNICh is a combined network interface card and datacenter switching accelerator. But unlike existing network interface-based solutions, the sNICh supports all datacenter switching functionalities. This enables protection, isolation, and allocation to be performed uniformly across the datacenter.

The sNICh architecture exploits the proximity of the switching hardware to the server by carefully dividing the network switching tasks between them. This decoupling of switching tasks enables the sNICh to address the resource intensiveness of exclusively software-based approaches and the scalability limits of exclusively NIC-based approaches. Thus it enables the delivery of complex switching functionalities needed in the datacenter. For example, while basic packet switching is implemented in hardware, packet filtering using access control lists (ACLs) is performed in software. But this division by itself is not useful if every packet has to traverse the software path. Instead, the sNICh architecture implements flow-based packet switching. So the flows are validated once in software and the validated flows are then cached in hardware. Thus the subsequent packets belonging to the validated flows can be completely handled in hardware.

Further, since the sNICh is aware of all the virtual machines (VMs) on the server, it can manage the network traffic on a per-VM basis. The sNICh also extends network QoS all the way to the VMs, and thus addresses the problem of ensuring end-to-end QoS. Finally, the sNICh is architected to minimize the I/O bus utilization by transferring, wherever possible, all the inter-VM traffic within the main memory.

Essentially, the sNICh architecture takes the Hyper-Switch architecture one step further by offloading the data plane of the virtual switch to the network interface hardware, and thereby, completely eliminating the need for driver domains. The rest of this chapter is organized as follows. Section 5.2 describes the sNICh architecture and how it takes advantage of its tight integration with the server. Section 5.3

describes the sNICh prototype that was built using software emulation. Section 5.4 presents a performance evaluation of the sNICh architecture using this prototype.

5.2 sNICh Architecture

The sNICh, as the name suggests, is a combination of a NIC and a switch. The sNICh acts as both an interface between the server and the external network and as a switch for the VMs within the server. Figure 5.1 illustrates sNICh's high-level architecture, and its relationship with the rest of the server and the greater datacenter network.

As far as the guest VMs and the hypervisor are concerned the sNICh looks like a direct I/O NIC. This avoids the need to move packets to/from a software intermediary, which can be expensive. So the sNICh presents a regular NIC-like interface to multiple VMs. These interfaces can be created and destroyed by the management software as needed.

A fundamental limitation of today's direct I/O NICs is that they only support a rudimentary switch with basic functionalities. It is not feasible, using a hardware-only approach, to incorporate the advanced switching functionalities in a NIC, without making it expensive. It is also challenging to implement all the features of an enterprise-class switch directly within a NIC and still support a large number of VMs. This is primarily because all common control plane functionalities of a switch have to be implemented entirely within the NIC. Moreover, packets destined for another VM on the same machine have to still traverse the I/O bus twice. This not only wastes I/O bus bandwidth but also affects external network traffic.

The sNICh solution overcomes these limitations to implement a full-fledged switch while enabling a low cost NIC solution, by exploiting its tight integration with the

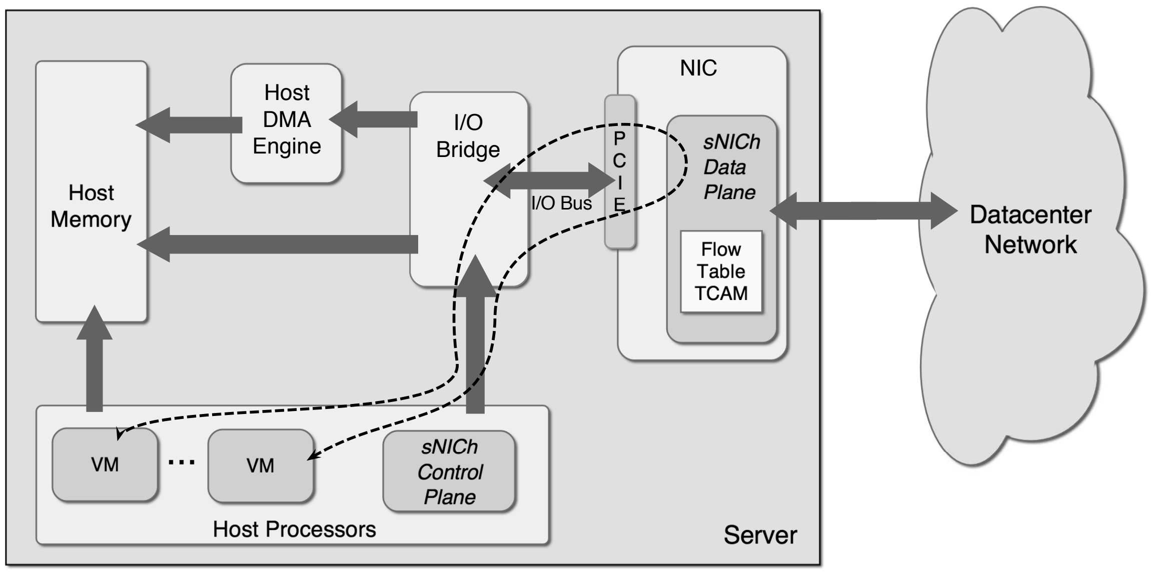


Figure 5.1 : *sNICh's high-level architecture and its relationship with the rest of the server and datacenter. The sNICh hardware implements the data plane of a flow-based switch. It also has a flow table TCAM to cache flow rules. The sNICh control plane is implemented in software. This architecture also takes advantage of the DMA engine on the host side to accelerate inter-VM traffic. As the dashed arrow shows, packet switching happens entirely within the server.*

server internals. This makes sNICh more valuable than simply a combination of a network interface and a datacenter switch. The sNICh architecture diverges from a conventional switch-on-the-NIC architecture in three ways. First, it separates the control and data planes in the last-hop switch. Whereas the data plane is implemented in hardware within the NIC, the control plane is implemented in host software. Second, it supports flow-based packet switching to ensure that the software path is not traversed on every packet. Finally, it takes advantage of DMA engines on the host-side to avoid wasting I/O bus bandwidth. The rest of this section explains the sNICh architecture in greater detail.

5.2.1 Separation of Control and Data Planes

A typical datacenter switch implements two components: a control/management plane and a data plane. The control/management plane performs address learning to figure how to forward packets and adds the results to a forwarding table or a forwarding information base (FIB). It also implements various management API(s) used to configure the switches.

The data plane performs packet forwarding, which involves parsing the packet headers, looking up the FIB, and transferring the packet to the right output port(s). Many switches, especially those used in datacenters, support more advanced functionalities in the data plane such as packet filtering. Packet filtering is necessary for functions such as access control list (ACL) processing and supporting QoS. These packet filtering operations typically involve multiple hash table lookups on simple regular expressions. In modern datacenter switches, this is performed using large ternary content addressable memories (TCAMs). TCAMs facilitate high-throughput pattern matching on selected fields in the packet header. However, TCAMs are expensive and extremely power-hungry. Therefore, a typical datacenter (top of the rack) switch is provisioned with a TCAM that can only support 1K-8K entries [61, 62].

In the sNICh architecture, the two components are separated by implementing the data plane within the NIC hardware and the control/management plane in software in a management domain[16]. This can be done efficiently due to the proximity of the sNICh hardware to the server.

[16]The management domain can be a separate VM like Xen's domain 0 or the hypervisor itself.

But it is not feasible to support conventional packet filtering in the data plane within the NIC hardware since it necessitates large TCAMs to accommodate all the ACL rules. So the only alternative is to maintain the ACL rules in the host memory and perform the filtering in software. In fact, even a software switch has to filter packets in the driver domain or the hypervisor, which typically involves list traversal and perhaps some limited hashing. Since most modern systems use TCP segmentation offload (TSO), such packet filtering in software can be done on large packets, instead of MTU-sized packets. This can decrease the filtering burden on the software by an order of magnitude. Even so, performing the necessary lookups in software, on every packet, is prohibitively expensive. It is not possible to approach the filtering throughput of the dedicated TCAM hardware in a datacenter switch.

So conventional packet filtering is not performed in the sNICh architecture. Instead, new packet flows are *validated* against the ACL rules in software. Then the validated flows are cached in hardware. The subsequent packets belonging to these validated flows are handled entirely in hardware and therefore, do not incur the software filtering costs.

Another benefit of implementing the control/management plane in software is that it can easily support the necessary management API(s) to enable uniform configuration of all switches across the datacenter. In fact, today's software switches already support such management solutions. The sNICh architecture can be a part of a similar datacenter wide network management solution. For instance, OpenFlow [63] can be used to manage the switches in the sNICh architecture. The sNICh control plane can "speak" the OpenFlow protocol [64] to a centralized external controller, which can then configure and manage the switches.

Therefore, by implementing the control/management plane in software, not only is the complexity of the sNICh hardware reduced, but it also affords the flexibility of supporting a multitude of features.

5.2.2 Flow-based Packet Switching

The communication between the control and the data planes, in the sNICh architecture, is implemented using a flow-based interface. A packet is switched in the data plane in three steps:

- **Flow identification:** The packet header fields are parsed to identify the corresponding packet flow. These header fields include the source MAC address, the destination MAC address, the Ethernet type, the VLAN id, the source IP address, the destination IP address, the transport protocol number, the source port number and the destination port number[17].

- **Flow table lookup:** The packet flow is used to lookup a matching flow rule in the flow table. The flow table is implemented using a TCAM in hardware. Ideally, the TCAM size should be large enough to accommodate most of the active flow rules and small enough to be implemented inside the sNICh hardware. When a flow table lookup fails (*a cache miss*), the packet is forwarded to the sNICh control plane software.

- **Flow action execution:** A successful flow table lookup (*a cache hit*) identifies a flow rule, which specifies the action to be performed. Typically, the action is to forward the packet to a one or more output ports or to drop the packet. But it may also include advanced actions like flow classification and prioritization.

[17]The flow is defined here only for a TCP/UDP and IP packet. This can easily be extended to accommodate other transport and network protocols.

Thus the flow actions can be used to implement access control and QoS in the sNICh architecture.

A packet that is switched entirely in hardware represents the best-case switching scenario. But packets may have to be handled in software when a flow table lookup fails (*a cache miss*). The lookup can fail due to two reasons, either the packet belongs to a new flow or the flow rule is no longer cached in hardware. Then the sNICh control plane software is responsible for correctly forwarding the packet.

When a packet reaches the sNICh control plane, a new flow rule is created along with an action to forward the packet to the appropriate output port(s). A new flow rule is *synthesized* from the entries in the FIB, the ACL rules, and any other flow classification/prioritization information. Then the new flow rule is cached in the hardware flow table. If the flow table is full, then the least recently used (LRU) flow rule is replaced. Finally, the packet is re-injected into the sNICh hardware. So the sNICh control plane maintains all the flow rules and caches them in hardware as needed. If the packet belongs to an existing flow, then the corresponding flow rule, which has already been synthesized, can be directly cached in the hardware.

Packet validation: A packet belonging to a new flow is validated against all the ACL rules. Access control rules can be of two types. In the first type, the default action is to block traffic and the ACL rules specify the flows that should be allowed. If a packet matches such an ACL rule then it is processed normally. Otherwise, a "negative" flow rule is created based on the packet's headers and the rule is cached in hardware. Now all subsequent packets that belong to this flow will be dropped in the hardware itself.

In the second type, the default action is to allow traffic and the ACL rules specify flows that should be blocked. If a packet does not match any of these rules, then it is processed normally. But if a packet matches one of these ACL rules, then a negative flow rule based on the ACL entry is created and cached in hardware. Now all subsequent flows that match this ACL rule will be dropped in the hardware itself.

The negative flows are cached in hardware only when a packet belonging to an active flow is blocked by the ACL rules. Thus unnecessary utilization of the flow table TCAM, which is a scarce resource, was avoided.

While there have been significant advances in algorithmic solutions for packet classification, fast packet classification in software is still a challenge [65]. TCAMs still remain the favored solution for implementing high-throughput pattern matching using wildcard rules. Typical ACL rules make heavy use of wildcards. For example, an ACL rule that blocks all traffic to a particular port has the IP source and destination fields, among others, wildcarded. The sNICh can then cache this rule as a negative flow rule in hardware and have the packets dropped in hardware itself. So a malicious guest VM that tries to overwhelm the control software by sending packets belonging to different flows, say different TCP/IP addresses and ports, can be stopped, since a small number of TCAM flow rules can effectively block all the illegitimate packets. While this is not proof that sNICh can block all malicious traffic, it does show the value of efficient wildcard matching for ACL rules.

5.2.3 Offloading Packet Copies to Host-Side DMA Engines

After a direct I/O NIC has determined if the destination of a packet is within or outside the server, the packet must be transferred to the output port. If the destination is outside the server, then the packet is transmitted over the external link. However, if the packet is destined for a VM co-located on the same machine (or even to the control plane software in the management VM), then the packet has to be copied, via DMA, back to the main memory on the host side of the I/O bus. In other words, every internal packet[18] is forced to traverse the I/O bus twice, before and after the packet is switched in hardware. This is the typical mode of operation in a conventional direct I/O NIC. However, this can result in an inefficient use of the I/O bus bandwidth.

In contrast, in the sNICh architecture, *only the packet headers are initially copied*, via DMA, to the hardware. These packet headers are then used to perform the switching operations. If the destination is outside the server, the rest of the packet is then copied using DMA and the packet is transmitted over the external link. However, if it is an internal packet, then it is directly copied from the source to the destination VM.

Modern server platforms enable offloading of data copies using DMA engines. The DMA engines, typically built into the server chipset, can perform asynchronous memory-to-memory movement of data, thus freeing up CPU cycles to perform other compute tasks. The sNICh architecture takes advantage of the DMA engines to perform asynchronous packet copies from the source to the destination VM while switching internal packets. This is feasible due to the proximity of the sNICh hardware to the server internals.

Server DMA Architecture

The DMA engine on modern Intel platforms [66] is used as an example here to describe the steps involved in setting up asynchronous DMA operations. The DMA engine presents a standard PCI-E device interface. It supports multiple independent

[18]A packet that has to be forwarded to control plane software in the management VM is also referred to as an internal packet.

DMA channels to the host memory. Each DMA channel has a queue of pending asynchronous transactions associated with it, where a transaction describes the operation to be offloaded to the DMA engine. A transaction is setup by creating hardware descriptors in the host memory. Each hardware descriptor includes the source physical address, the destination physical address, and the size of the data to be copied. The transaction descriptors queued for a particular DMA channel are linked together and the physical location of the first descriptor is given to the DMA engine. Once the new transactions are setup, they are pushed to the DMA engine by tickling one of its registers. This is also sometimes called a “door bell”, since it triggers the processing of the pending transactions. Then the DMA engine performs the asynchronous data copies without any processor intervention.

Transaction completion can be detected either by polling or using interrupts. Polling is done by reading the value in a pre-configured completion writeback area in the host memory. This indicates the last completed transaction. Since the transactions are processed in-order, the last completed transaction indicates all the transactions that have been completed. Alternatively, each channel in the DMA engine can be configured to deliver MSI-X interrupts to the host processor on completion of the asynchronous transactions.

sNICh DMA Architecture

The sNICh’s proximity to the host enables it to exploit a DMA engine within the server to accelerate packet copying operations. The sNICh can effectively setup a DMA operation, independent of the host, to enable packet copying in the background. So the copying be performed without any software intervention and the wasting of I/O bus bandwidth.

In general, it has been shown that utilizing such DMA engines is only profitable for large copies, as the setup and completion overheads outweigh the benefits [59]. The setup costs arise largely from writes to uncached memory regions to store the DMA descriptors. The completion costs arise largely from polling or interrupt overheads.

The sNICh avoids these overheads in two ways. First, the descriptors are stored in memory, via DMA, from the sNICh hardware. Therefore, the sNICh hardware can perform these writes in the background and in parallel with other operations, whereas the CPU would likely stall waiting for the memory operations to complete. Second, the sNICh can easily mitigate the completion overhead by limited polling. After copying a packet using the DMA engine, the sNICh hardware notifies the receiving VM that a new packet has been received. As NICs use interrupt moderation to batch notifications to the operating system, the sNICh hardware need only poll the DMA engine for completion once every interrupt moderation period, which is frequently $100\mu s$ or more on modern high performance NICs. Therefore, the sNICh architecture provides an extremely effective way of exploiting the server DMA engines to accelerate packet copying among VMs.

While not a technical challenge, current server platforms do not allow PCI-E devices to access each other's memory spaces. But peer-to-peer PCI-E messages are supported in the PCI-E specification and are expected to be included in future server platforms.

5.3 sNICh Prototype

We built a prototype of the sNICh architecture, where all the hardware components were emulated in software. The prototype comprised of three components: 1) the sNICh data plane emulation, 2) the sNICh control plane software, and 3) the sNICh driver in the guest VMs. We used Xen [24], an open source virtualization platform, as our testbed. The prototype only supported inter-VM packet switching. In other words, guest VMs could not send and receive packets to and from external networks using this prototype. Further, the used of the DMA engine to offload packet copies was not emulated in this prototype. Instead, standard copies replaced any DMA operation.

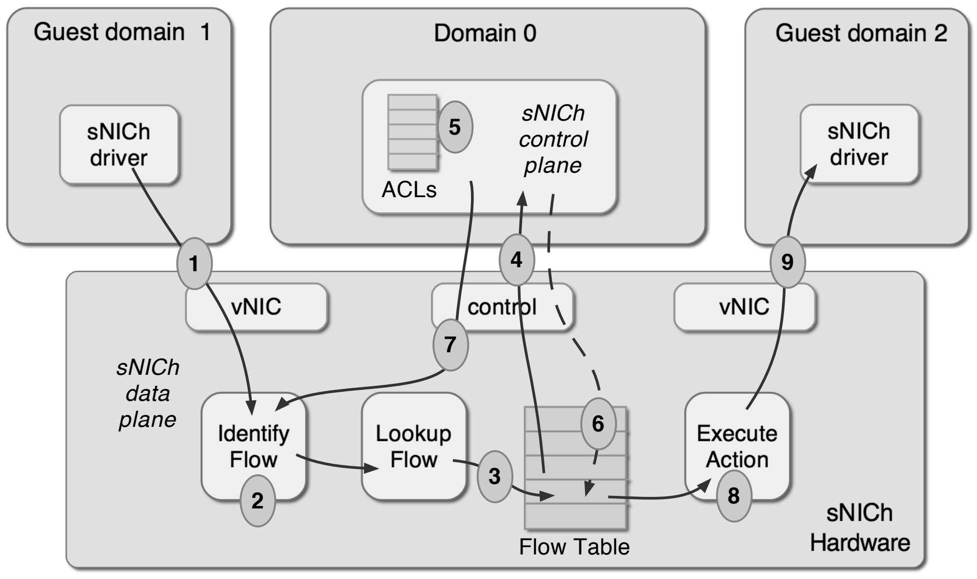
The sNICh hardware emulation that implemented the data plane was run on a dedicated processor core to isolate it from the host software. It contained a single thread that constantly monitored all the virtual network interfaces (vNICs), in a round-robin fashion, for packets to be switched. The sNICh flow table was emulated using hash tables in software (in place of hardware TCAMs). The maximum number of flow rules in the flow table was limited to 1024, to emulate TCAM restrictions. Hardware interrupts were emulated using inter-processor interrupts (IPIs). So a notification was delivered by first sending an IPI to the appropriate processor core and subsequently, the hypervisor delivered a virtual interrupt to the recipient software component.

The sNICh control plane was implemented in Xen Linux domain 0 (Xen's management VM) and the sNICh driver in Xen Linux DomU (a para-virtualized (PV) guest domain). The sNICh control plane was initialized when domain 0 booted up. This also set up the control interface, which was used for the communication between the emulated sNICh hardware and the sNICh control plane software. The control interface contained a pair of command descriptor rings for communication in either direction. The descriptor rings were implemented using shared-memory. The packets were communicated out-of-band using memory-copies, which replaced the standard DMA operations in the emulation. The descriptor rings were also used by the control plane to install and uninstall flow rules in the emulated hardware flow table. The flow rule description was also communicated out-of-band to the sNICh hardware. For example, the control plane installed a new flow rule by storing the flow description in a buffer in the host memory. Then a new command descriptor, which pointed to this buffer in memory, was added to the descriptor ring. Finally, the ring producer index was updated. This triggered the processing on the emulated sNICh hardware side, which copied the flow description and added a corresponding flow rule to the emulated flow table. ACL rules were installed in the control plane software using a user-level tool. Packet flows were validated against the ACL rules using a simple linear search algorithm.

When a guest domain booted up, it registered with the sNICh control plane. During registration, the sNICh control plane was informed of the MAC address allotted to that VM. Then the control plane created a new vNIC interface on the sNICh hardware. The vNIC interface implemented a pair of descriptor rings for packet transmission and reception. The implementation of the descriptor rings was similar to that of the command descriptor rings in the control interface described earlier. The guest VM and its vNIC interface were also bound to a unique virtual port (vPort) on the sNICh hardware. Once the registration was completed, the guest VM could directly send and receive packets to and from other guest VMs via the emulated sNICh hardware.

The following steps are involved in switching and forwarding a packet between two guest VMs through sNICh. The figure does not show the use of DMA engines.

1. The sNICh driver in domain 1 transmits the packet to the sNICh hardware by queuing it at its vNIC interface.



1 Packet transmitted by domain 1.
2 Flow idenfitied.
3 Flow table looked up.
4 Packet forwarded to control plane.
5 Flow validated and flow rule synthesized.
6 New flow rule installed.
7 Packet re-injected by control plane.
8 Flow action executed.
9 Packet delivered to domain 2.

Figure 5.2 : *Steps involved in switching a packet between two guest VMs through sNICh.*

2. The sNICh hardware then copies the packet headers from the host memory and identifies the corresponding flow by parsing the packet's headers.

3. The flow table is looked up for a matching flow rule.

4. If the lookup fails, then the packet is sent to the sNICh control plane software through the control interface. The DMA engine is setup to copy the contents of the packet from domain 1 to the control plane.

5. If the packet belongs to a new flow, then it is validated against the ACL rules

and a new flow rule is synthesized.

6. The flow rule is cached in hardware through the control interface.
7. The packet is re-injected into the sNICh hardware again through the control interface.
8. If the flow table lookup is successful, the corresponding flow action is executed.
9. The packet is forwarded to domain 2 through its vNIC interface. The DMA engine is used again to copy the contents of the packet to domain 2.

Steps 4-7 are not needed when the flow table lookup is successful.

5.4 Evaluation

This section presents a performance evaluation of the sNICh architecture using the sNICh prototype. In this evaluation, the performance of the sNICh architecture was compared with the performance of two software switching solutions in Xen, the *Linux Ethernet bridge* and the *Open vSwitch*. Traditionally, Xen has used the driver domain model to support I/O virtualization [8]. The driver domain also implements a software switch. So the Xen guest domains send all network packets to the driver domain and the driver domain then switches the packets to forward them to another guest domain or to the external network.

The Linux bridge is a software Ethernet switch which is shipped with the Linux kernel [18]. The netfilter/iptables framework [67] is used in the Linux bridge to support ACLs. Open vSwitch [36] is an OpenFlow [63] compatible, open source software switch. OpenFlow uses a flow-based approach to switch packets, similar to the sNICh architecture proposed in this thesis. Open vSwitch implements two network paths: an in-kernel fast-path and a user-level slow-path. The fast-path is implemented as a kernel module which replaces the Linux Ethernet bridge in the driver domain. In the fast-path, a software flow table is looked up for a matching flow rule and the actions associated with it are executed. When the flow table lookup fails, the packets are forwarded to the user-level control software via the slow-path. Open vSwitch also provides a tool, called `ovs-ofctl`, to install ACL rules.

5.4.1 Experimental Setup and Methodology

The netperf UDP stream microbenchmark [45] was used in all the experiments to generate network traffic between Xen guest domains. Five UDP packet sizes (250, 450, 650, 850, and 1050 bytes) were used in the experiments. The packet throughput at the switch was used as the metric to compare performance. The rate at which a switch processed the packets was limited either by the CPU on which the switch was running or the CPU at the transmit side guest domain. It was also ensured that there were no packet drops at the switch to avoid distorting the throughput calculations. Further, the CPU cost metric was used to study the performance overheads in the experiments. The CPU cost was measured as the CPU cycles consumed per packet (henceforth referred to as *cycles/packet*). The cycles/packet metric was computed by dividing the total packet processing cost in the driver domain across the total number of packets processed at the switch.

The experiments were run on an Intel server machine with a 2.67 GHz Intel Xeon W3520 quad-core CPU and 6 GB of memory. The server ran Xen and was configured with up to three guest domains, each with a single virtual CPU (vCPU) and 1 GB of memory. Further, each vCPU was pinned to a separate physical CPU core to eliminate potential issues related to scheduling domains and I/O performance [48].

The final processor core was used to run either the sNICh emulation or the driver domain. The driver domain was configured similar to the guest domains.

5.4.2 Experimental Results

Figure 5.3 compares the packet throughput at the three switches when there were no ACL rules to process. It was observed that the sNICh out-performed both the software switches for all packet sizes. The throughput at both the Linux bridge and the Open vSwitch was limited by the driver domain's CPU. But in the sNICh architecture, the throughput was limited only by the CPU at the transmit side guest domain. Consequently, the sNICh was able to process more packets and hence achieved better performance.

Figure 5.4 shows the CPU cost incurred in the driver domain when processing 650 byte packets. This cost has been divided into three categories—the packet switching, the packet copying, and the misc overheads. The *misc* overheads essentially represent the cost of interfacing with the driver domain to switch packets. This primarily included the cost incurred in Xen's network backend driver and in Xen's memory sharing mechanism. This cost dominated the total packet processing cost in the driver domain. For instance, this accounted for ∼88% of the total CPU cost when using the Linux bridge.

In these experiments, the performance difference between the software solutions and the sNICh was primarily due to the cost incurred in moving packets between the guest domain and the driver domain. This cost was not incurred in the sNICh architecture since the sNICh hardware is essentially a direct I/O NIC and the packets do not traverse the driver domain software. For the same reason, the overhead of Xen's memory sharing mechanism was also completely avoided. The cost incurred in the

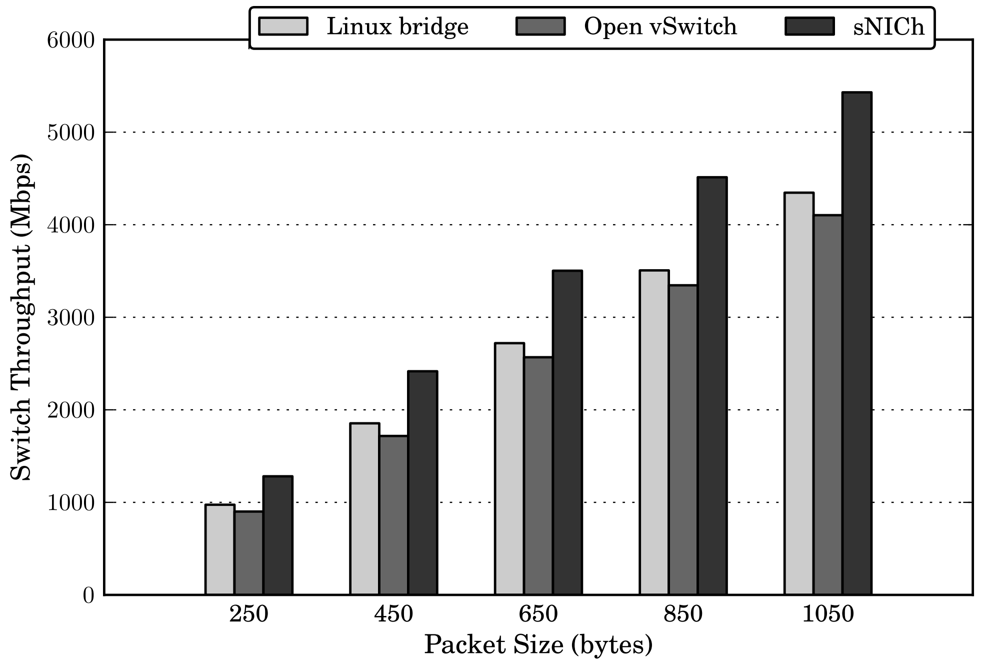


Figure 5.3 : *Packet throughput at switch without packet filtering*

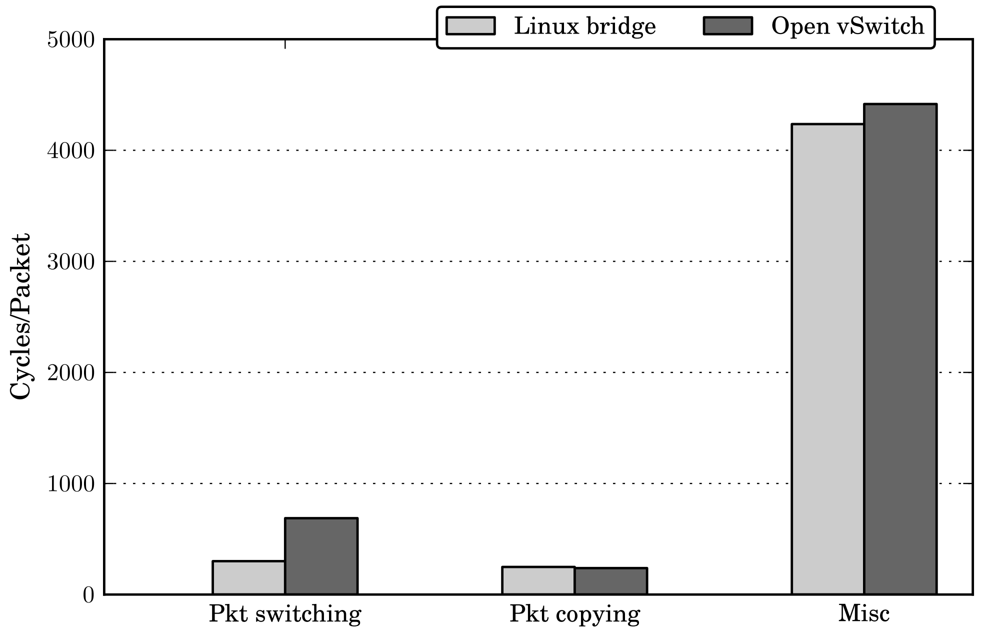


Figure 5.4 : *Packet processing cost in driver domain without packet filtering*

emulated sNICh hardware has not been included here since we are only comparing the host CPU overhead.

Figure 5.5 compares the packet throughput when packet filtering using ACLs was enabled at the switches. In these experiments, the packets were filtered using a single ACL with 971 reject rules. The ACL rules were generated using the ClassBench toolkit [68]. Each ACL rule filtered packets based on five packet header fields—the source and destination IP addresses, the source and destination ports, and the protocol number, or some subset of these fields. But none of the generated packets matched any of these rules. This scenario was chosen, since in most systems, the number of packets dropped by the filter is much smaller than the number of packets that get admitted. Here it was observed that the throughput at the Linux bridge was reduced by up to 29% as compared with the previous set of experiments (Figure 5.3). But the performance of both the flow-based switches were similar to their performance in the experiments without ACL processing.

Figure 5.6 shows the CPU cost in the driver domain when processing 650 byte packets. This cost has been divided into four categories—the packet switching, the packet filtering, the packet copying, and the misc overheads. Under the Linux bridge, the packet filtering operation consumed ∼75% of the total CPU cost and thus, dominated the processing costs in the driver domain. This was due the ACL implementation using the netfilter/iptables framework in the Linux bridge where every packet is serially compared against all the ACL rules. But under the Open vSwitch, the filtering costs were negligible. In general, the performance of the flow-based switches were not affected since only the first few packets of the flow incurred the filtering cost, until the flow was validated against the ACL rules and a new flow rule was installed in the flow table.

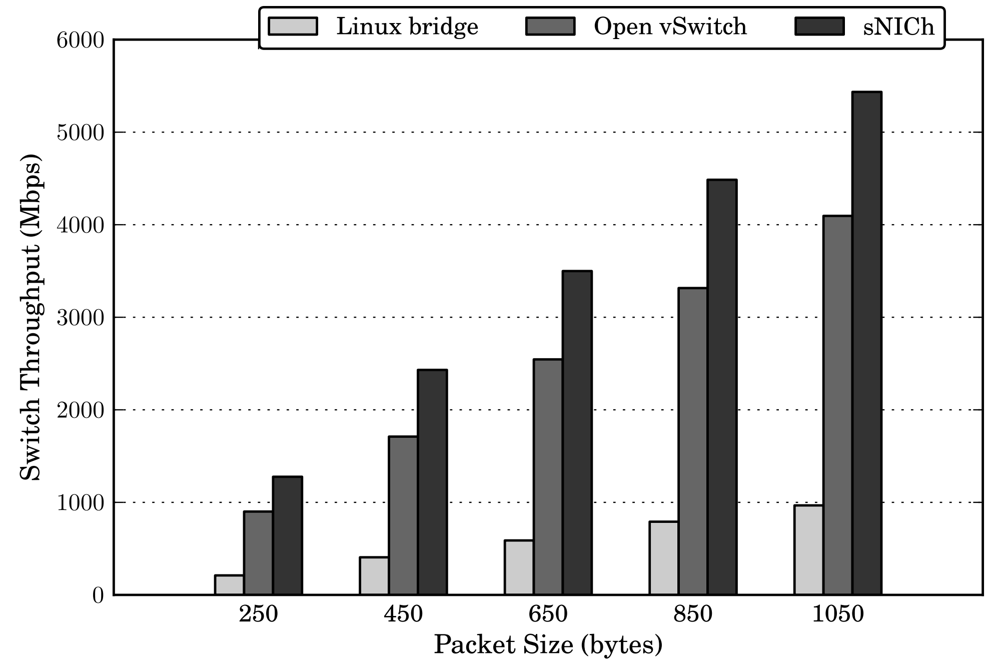


Figure 5.5 : *Packet throughput at switch with packet filtering*

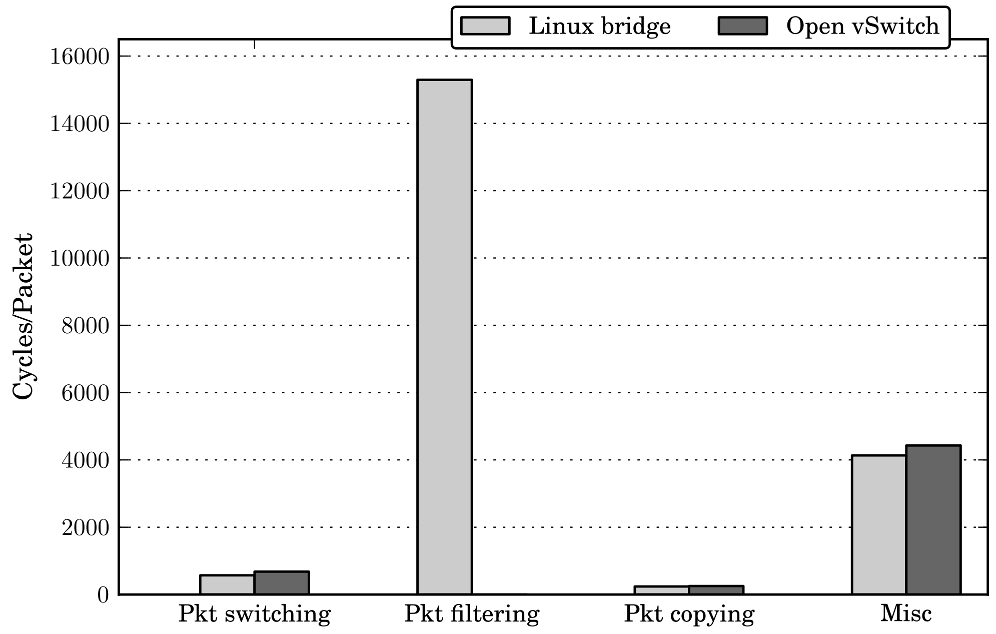


Figure 5.6 : *Packet processing cost in driver domain with packet filtering*

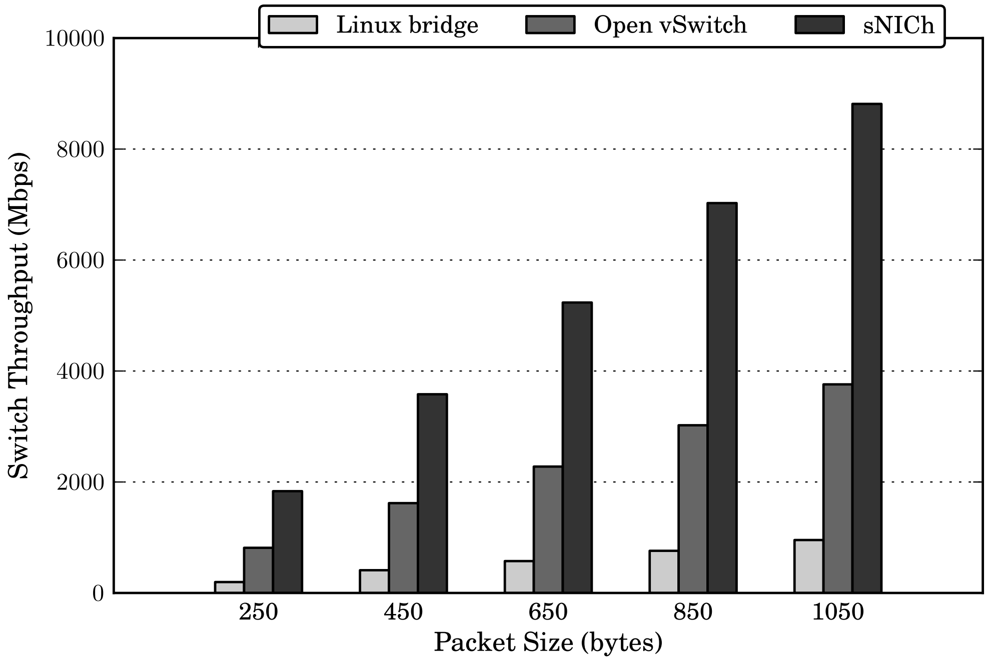


Figure 5.7 : *Packet throughput at switch with 2 TX guest VMs (with packet filtering)*

Figure 5.7 compares the packet throughput at the switches when there were two guest domains transmitting packets. Under the sNICh architecture, the throughput nearly doubled for all packets sizes as compared to the previous set of experiments where there was just one guest domain transmitting packets (Figure 5.5). But the performance of both the Linux bridge and the Open vSwitch did not scale. This was because, unlike under the sNICh architecture, the CPU at the switch in the driver domain was the performance bottleneck. Therefore, the sNICh exhibited better scalability than both the software solutions.

5.5 Conclusions

There are inherent overheads in any software-based solution for device virtualization and last hop switching. There are alternate proposals to send all network packets to external switches, which inherently waste link bandwidth. A nice middle ground in this space is to switch packets within the server's network interface cards (NICs). But it is not possible to implement a full-fledged switch inside a NIC without making it complex and expensive. This challenge is addressed by introducing the concept and design of a new I/O subsystem called the *sNICh*, which is a combination of a network interface and a switching accelerator.

The sNICh utilizes minimal and off-the-shelf building blocks to support key elements of data center switching and leverages its proximity to the server to provide efficient last-hop data center switching. This makes sNICh more valuable than simply a combination of a network interface and a datacenter switch. The sNICh architecture diverges from a conventional switch-on-the-NIC architecture in three ways. First, it separates the control and data planes in the last-hop switch. Second, it supports flow-based packet switching to ensure that the software path is not traversed on every packet. Finally, it takes advantage of DMA engines on the host-side to avoid wasting I/O bus bandwidth.

A prototype of this architecture was built using software emulation to evaluate this solution. It was shown to outperform and scale better than the existing solutions.

Chapter 6

Related Work

While processor and memory virtualization have come a long way in terms of reducing the associated overheads and improving performance, the virtualization of devices, especially network devices, has lagged behind. Until recently, there has not been much interest in the industry to explore new I/O virtualization architectures. However, there has been a lot of work in the research community to overcome the challenges posed by I/O virtualization. This chapter discusses some of the prior research that closely relate to the contributions of this thesis. It is organized as follows. Section 6.1 describes various proposals to reduce the software I/O virtualization overheads to improve network performance. Section 6.2 presents the direct device access schemes that eliminate the software overheads during I/O virtualization. Section 6.3 explains the previous efforts to optimize memory sharing during I/O operations. Section 6.4 discusses the current state-of-the-art solutions to support last hop virtual switching in datacenters. Section 6.5 describes various ways in which network processing has been distributed, offloaded, and onloaded in systems. Section 6.6 explains the previous uses of flow-based packet switching and finally, Section 6.7 includes some miscellaneous related work.

6.1 Reducing Software I/O Virtualization Overheads

Reducing the software I/O virtualization overheads has been the focus of an extensive body of work. Initially, the hypervisor emulated all I/O devices in software [69]. But this resulted in very poor performance since the hypervisor had to trap all I/O operations by guest VMs and then convert them into operations on the real device, which could potentially be different from the emulated device. So para-virtualization (PV) was introduced where guest VMs co-operated with the hypervisor through a standard device interface using virtualization-aware drivers [8, 70, 71]. This reduced the costs associated with virtualizing devices in software. However, despite this reduction, the gap in network I/O performance between native and virtualized systems was still significant.

Menon *et al.* [72] added features such as scatter-gather I/O, TCP/IP checksum offload, and TCP segmentation offload (TSO) to virtual network interfaces to reduce the packet processing overheads in Xen's network path. Further, they also advocated re-introducing packet copies, in place of page flipping, in the network receive path. The packet copies are needed to move incoming packets from the driver domain to the guest VMs. They showed that copying the contents of the packets, especially small packets, into the guest I/O buffers could be more efficient than transferring the pages containing the packets to the guest VMs.

Santos *et al.* [41] proposed moving the packet copies from the driver domain to the guest VMs. In other words, the guest VMs copied the contents of the incoming packets from buffers in the driver domain, as opposed to the drive domain copying the packet contents into buffers in the guest VMs. They argued that performing the packet copies in the guest VMs allowed for better utilization of the CPU caches. In particular, it sped up the subsequent copy of the packets' contents into application buffers within the guest VMs. Further, it also avoided polluting the driver domain's CPU caches. But recent architectural features such as direct cache access (DCA) could make these optimizations redundant [73].

Liao *et al.* [52] proposed an alternate means to reduce the inter-domain packet copy costs. They designed a new CPU cache-aware scheduler that attempted to take advantage of CPU cache locality to accelerate the packet copies. Essentially, their scheduler used a heuristic, based on the number of I/O events, to locate the vCPUs that corresponded to driver and guest domain pairs and scheduled each of them to run on processor cores that shared a CPU cache. But their methodology could break down when two or more guest domains used the driver domain to concurrently send or receive network packets. The Hyper-Switch architecture presented in this thesis is also optimized to take advantage of CPU cache locality when offloading the packet processing to idle processor cores. The Hyper-Switch architecture can better utilize the CPU caches since it has precise knowledge of the processor cores on which the receiving VMs are running.

But packet copying still incurred significant costs in the network receive path [41]. Fundamentally, packet copying was required since packet de-multiplexing was performed in software *after* the network interface cards (NIC) had DMAed the packet to the software intermediary (the driver domain or the hypervisor). Thus the copying overheads could be eliminated if the de-multiplexing was performed *before* the DMA operation. Modern multi-queue network interface cards provided this capability by de-multiplexing the incoming packets, in hardware, into different receive queues using the packet's destination MAC address. For instance, Intel's 82598 10 Gigabit Ethernet controller [74] supports this feature. This enabled multi-queue NICs to be used for virtualization where each network interface queue was dedicated to a specific guest VM [34, 37]. These network cards were able to identify the target guest VM for all incoming network traffic by associating a unique Ethernet MAC address with each receive queue. Then they de-multiplexed the incoming network traffic based on the packet's destination Ethernet MAC address to place it in the appropriate receive queue. However, since the number of receive queues were limited, only a small number of guest VMs could take advantage of this feature. The rest had to fall back on a shared receive queue. Further, this solution also complicated packet filtering since the filtering was typically performed in the driver domain before the packets were delivered to the destination guest VMs.

Finally, Ram *et al.* [34] studied the impact of several virtual driver optimizations that reduced the I/O virtualization overheads in a guest VMs' network receive path. These optimizations included large receive offload (LRO), software prefetching, and half page buffer allocation. LRO aggregated the arriving TCP/IP packets into a smaller number of larger-sized packets (TCP segments), and then passed the large segments to the network stack. As a result, the network stack processed a group of packets as a single unit at the same cost as processing a single packet. Software prefetching reduced the cache miss penalty by prefetching the packet data headers and the socket buffer data structures. The half page buffer allocation optimization advocated the use of smaller receive buffers by guest VMs to reduce their memory footprint.

6.2 Direct Device Access Proposals

Another category of work investigated the elimination of the software overheads by moving the virtualization support to the network interface card itself. These solutions were also referred to as *direct I/O* or *device pass-through*. Initially, there were several such design proposals from the academia to allow guest VMs to directly access the network card bypassing any software intermediary [12–16]. For instance, Willmann *et al.* [13] presented their Concurrent Direct Network Access (CDNA) architecture where each guest VM was presented a unique *virtual context* on the network device. This allowed the guest VMs to safely and directly communicate with the network device without compromising the isolation and protection of VMs. These solutions achieved near-native network I/O performance since the only source of virtualization overheads was the costs involved in handling device interrupts in the hypervisor and converting them into virtual interrupts for the target guest VMs.

Eventually, the direct device access proposals led to an industry-wide standard called Single Root I/O Virtualization (SR-IOV) [27]. The SR-IOV standard, developed by the PCI-SIG consortium, extended the PCI-E specifications to provide a standard mechanism for a network device to be natively shared by multiple guest VMs. In particular, it introduced the concept of virtual functions (VFs) to logically partition PCI-E devices. The virtual functions were provisioned with a carefully selected, minimal set of resources to allow just basic data movement between the device and the guest OS [75]. Today, the SR-IOV standard is being rapidly adopted in the industry, with several network interface vendors supporting this feature in their products [28–30].

In the sNICh architecture presented in this thesis, the sNICh hardware looks very much like a direct access network card to the guest VMs and the hypervisor. So the guest VM's can directly communicate with the sNICh hardware bypassing any software intermediary. But, unlike existing direct access network devices, the sNICh supports all the datacenter switching functionalities by exploiting the proximity of the switching hardware to the server.

However, unlike virtualizing devices in software, direct device access cannot provide fault isolation and device transparency. In particular, it requires device-specific code in the guest VM, which has several negative consequences. It increases guest image complexity, reduces guest VM portability, and complicates live guest migration [31–33] between systems with different devices. Moreover, the devices that support this feature provide just a limited number of virtual interfaces. Therefore, scalability is also a concern. For these reasons, despite the performance gap, software-based I/O virtualization still remains the most popular solution.

6.3 Memory Sharing During I/O

As described in Chapter 3, in I/O virtualization architectures that use driver domains, memory sharing during I/O operations occurs in two levels. First, I/O buffers in guest domains' memory are shared with the virtual network devices implemented in the driver domain. Second, the I/O buffers are shared with the physical network devices. There has been prior work on mechanisms used to support both the levels of sharing.

In the Xen virtualization platform, the first level of memory sharing between VMs is supported using the grant mechanism. But the grant mechanism incurred significant overheads when performing network I/O operations. Therefore, Santos *et al.* [41] proposed a reuse scheme for the Xen's grant mechanism to reduce memory sharing overheads. But their work neither implemented the scheme nor specified a design, but merely estimated the potential benefits by running experiments with the grant mechanism disabled. Subsequently, Ram *et al.* [34] designed an initial version of the grant reuse scheme. But this version suffered from several limitations. It supported the reuse scheme only for network traffic on the receive path, and further, only for network cards with multi-queue support. Specifically, it lacked the ability to map guest VMs' memory pages into the driver domain's memory and to reuse the mapped pages across multiple I/O operations. In essence, it could only be used by a guest domain to grant a driver domain the access to program DMA operations to the guest VMs' memory pages. In contrast, the new memory sharing mechanism presented in this thesis supports unilateral revocations. This in turn makes it simple for a guest OS to implement any reuse scheme, which could be used under any network virtualization scenario.

The second level of memory sharing is supported using *I/O Memory Management Units (IOMMUs)* [38, 39]. Originally, IOMMUs were used to handle disparities in addressing between the host processor and the I/O devices [40]. Typically, the devices could address only a small portion of the physical address space of the host processor. This became an issue especially with the advent of 64 bit processors. So IOMMUs were used to translate the smaller device addresses into larger physical addresses. Therefore, IOMMUs were primarily used for address translation capabilities than for protection.

In a virtualized environment, IOMMUs play an even bigger role. They are used enforce memory isolation between VMs. Without IOMMUs, it is possible for a malicious device or an errant DMA to corrupt the memory of guest VMs or even the hypervisor. The need for IOMMUs is even more important with direct device access. In this case, a guest VM has direct access to the device to setup DMA operations. As a result, a malicious guest VM could easily program DMA operations to other guest VMs' memory through the device. Under these scenarios, the IOMMUs are useful for their protection capabilities as well. Intel [38] and AMD [39] have both added IOMMUs to their processor chipsets specifically for their use in virtualized systems.

Under Xen's traditional driver domain model, the IOMMU was setup during the grant operations. This provided fine-grained protection. However, under direct device access, the IOMMU was pre-configured with all the memory pages that belonged to the guest VM that was accessing the device. But this only provided coarse-grained protection. Early studies of IOMMU performance demonstrated a high cost of setting up/tearing down page mappings and performing IOTLB invalidations [42]. Thus incurring these overheads on every I/O operation became untenable since it severely limited the network bandwidth that could be supported. Subsequent work by Paul *et al.* [43] showed that the mappings could be reused to reduce the overheads associated with IOMMUs, but without necessarily resorting to the coarse-grained protection mode. In particular, they compared the performance of several protection strategies, including single-use, shared, persistent, and direct mappings. They showed close to 100% reuse of IOMMU mappings when using the persistent mapping strategy with a 128K limit. Further, Amit *et al.* [44] also discussed the trade-off between different mapping strategies such as asynchronous invalidation, deferred invalidation, and optimistic tear-down. The optimistic tear-down strategy waited for a configurable time period before it tore down a stale mapping. But if the mapping was reused in this period, the timer was reset. Moreover, they showed good performance using this scheme. This thesis presented a unified approach to reuse both grants and IOMMU mappings. Further, two reuse strategies were described. The first strategy involved just placing an upper bound on the number of mappings. The second strategy involved revoking a mapping as soon as a page was re-purposed to be used for non-I/O operations.

Mechanisms have also been proposed to support fast and cheap inter-domain communication using static shared memory channels between the communicating VMs.

XenSocket [76] provided a standard POSIX socket level API to establish the shared communication channel. However, applications had to be rewritten to take advantage of this new socket level API. *XWay* [77] was a similar socket level mechanism that bypassed the TCP/IP network stack in the guest VM. But unlike XenSockets, it provided full binary compatibility for applications that used the TCP sockets. Both of these mechanisms avoided the memory sharing overheads by setting up *static* shared memory channels during initialization and then transferring data over them. But the new memory sharing mechanism proposed in this thesis enables efficient *dynamic* sharing of memory.

Mechanisms for controlled memory sharing have been developed for environments distinct from server virtualization. For example, the network protocol Remote Direct Memory Access (RDMA) allowed hosts to "advertise" their memory buffers to enable remote hosts to read and/or write the memory [78]. The advertisement involved providing the remote hosts with access keys called *steering tags*, which bore a resemblance to the grant references used in the grant mechanism. Similar to the memory sharing mechanism described in this thesis, the RDMA protocol allowed the hosts that advertised memory to remove the access rights from the remote hosts by invalidating the corresponding steering tags.

6.4 Last Hop Virtual Switching

The current state-of-the-art last hop virtual switching solutions fall into one of three categories based on where the switching tasks are performed. The first category of systems implement the virtual switch entirely in software either within the hypervisor (e.g. KVM [11], VMware ESX server [10]) or a driver domain (e.g. Xen [8], Microsoft Hyper-V [9]). These systems include a simple network card that is virtualized in software. Today, this category of systems is most commonly used in virtualized servers since it offers a rich set of features, including security, isolation, and mobility. There are several software virtual switches—such as Linux bridge [18], VMware vswitch [17], Open vSwitch [36], etc.—that are used in these systems. In fact, Cisco and VMware have also made this last hop switch look and behave similarly to other switches in the datacenter [19]. Recently, Rizzo *et al.* [79] have also proposed a new software virtual switching solution based on their netmap API [80]. They use memory-mapped buffers to avoid data copies inside the host.

The Hyper-Switch architecture proposed in this thesis also falls in this category. However, unlike Hyper-Switch, all these existing systems implement the entire virtual switch within a single software domain—either the hypervisor or the driver domain. As a result, their performance suffers. But the optimizations that were implemented in the Hyper-Switch architecture are applicable to many of these solutions. Also, in this thesis, the Hyper-Switch's performance was only compared with the performance under Xen and KVM. A recent report from VMware has shown an impressive performance of 27 Gbps between two VMs running on their vSphere architecture [81]. Unfortunately, it is hard to compare this to Hyper-Switch's performance since the hardware platforms used in these evaluations are vastly different.

The second category of systems employ the more sophisticated direct access network devices that contain multiple *contexts* in hardware [13–15]. So these devices present a unique virtual network interface directly for each VM. These network cards also implement a virtual switch internally in hardware. However, today most of them only implement a rudimentary form of switch. While features like packet filtering using TCAMs are being added to some of these network cards [29], such solutions will neither be scalable nor cost-effective. Further, these network devices waste substantial I/O bandwidth while switching inter-VM network packets because they always transfer the full packet payload twice over the I/O bus, to and from the network card.

The sNICh architecture proposed in this thesis is also a network card-based last hop virtual switching solution. However, unlike the existing switch-on-the-NIC solutions, it implements a full-fledged switch while enabling a low cost NIC solution, by exploiting its tight integration with the server internals. This makes sNICh more valuable than simply a combination of a network interface and a datacenter switch. Luo *et al.* [82] propose offloading Open vSwitch's in-kernel data path to programmable network cards. Similarly, one can also imagine offloading Hyper-Switch's data plane to the network interface hardware. These solutions can enable high-performance since the VMs directly communicate with the network card. But, in general, this category of solutions lack the flexibility that pure software solutions offer.

The third category of switching solutions attempt to leverage the functionalities that already exist in today's datacenter switches. This approach uses an external switch for switching *all* network packets. Today, there are two competing standards to implement this approach—Virtual Ethernet Port Aggregator (VEPA) [20] and VN Tagging [21]. The main benefit from this category of solutions is that it simplifies management since all traffic from the server now transits a traditional switch and hence can be managed by a network manager system. But, fundamentally, this approach results in a wastage of network bandwidth since even packets from inter-VM traffic must travel all the way to the external switch and back again.

6.5 Distributing Network Packet Processing

Most of the early work into network subsystem architecture focused on accelerating and scaling the protocol processing by distributing it across the network interface hardware and the host operating system software. Makineni *et al.* [83] showed that the protocol processing in traditional network stacks could be too CPU and memory bandwidth intensive to scale to Gigabit speeds. Several designs that were collectively known as *offloading* approaches addressed this overhead by moving the protocol processing from the host CPU to the network interface hardware [84]. For instance, Kim *et al.* [85] presented their design and implementation of a TCP connection handoff scheme. They argued that full TCP offload was not feasible due to the limited processing capabilities and limited memory on the network card. So in their scheme, the operating system selected some of the TCP connections to be offloaded to the network interface card. The packets belonging to the offloaded connections were processed entirely within the network card. Then they were directly handed to the application, bypassing the network stack in the host operating system. The rest of the packets were processed normally by the operating system in software.

In spite of the performance benefits, offloading architectures were considered too inflexible and hard to evolve and maintain [84]. Hence, an alternative class of proposals, known as *onloading* approaches, retained the protocol processing in the host operating system, but supplied it with hardware support to make it more efficient and scalable. For instance, Schlansker *et al.* [86] proposed the coherence attached network interface card design to eliminate the interrupt overheads by enabling efficient polling. The *Receive Side Scaling (RSS)* technique attempted to make the protocol processing scalable to multiple CPU cores by sending all the packets of a flow to the same CPU [87]. Intel's I/O Acceleration Technology (AT) [66] included a DMA engine on the system chipset that could be used to eliminate buffer copies in software. The *Direct Cache Access (DCA)* scheme [73] reduced latency and saved memory bandwidth, by placing the incoming packets directly into the processor caches, instead of the main memory.

Similarly, network packet switching has also been offloaded to network interfaces. But all the existing proposals perform a full offload of network packet switching to the network interface hardware. The integration of switch and server functionality was considered in an early InfiniBand NIC, called *InifiniBridge* [88]. Although this network card integrated an InfiniBand switch, it was primarily meant to extend the old PCI (not the PCI-Express of today) and SCSI buses to form a Storage Area Network (SAN). RiceNIC [89] and NetFPGA [90] are two extensible network cards with on-board FPGAs that can be utilized to integrate switching functionality. However, implementing a full-fledged switch with large hardware TCAMs is not feasible on these platforms due to FPGA resource limitations. For example, in the OpenFlow switch implementation on the NetFPGA platform by Naous *et al.* [91], they used a combination of small on-chip TCAMs and large hash tables in the off-chip SRAM to implement flow tables. Therefore, their implementation could only support a small number of wildcard-based flow entries using the TCAMs. In fact, today, all network interfaces that support the SR-IOV feature also implement a network switch in hardware [28]. But they only support rudimentary switching functionalities.

In this thesis, similar ideas were used to offload or onload, depending on one's perspective, a part of the network packet processing in the last hop switch of virtualized servers. The proposed architecture exploited the proximity of the switching hardware to the server by carefully dividing the network switching tasks between the host software and the network interface hardware. Some of the processing (the data plane) was offloaded to the network interface hardware from a purely software-based solution or some of the processing (the control plane) was onloaded to the host software from a purely network interface-based solution.

There have also been several proposals to dedicate processor cores to perform packet processing. For instance, Regneir *et al.* [92] designated one or more of the CPU cores as packet processing engines (PPEs) that were used exclusively for protocol processing on the network stack. A shared memory interface was used for the communication between the host CPUs and the PPEs, which were then used to queue packet processing requests.

In a virtualized setting, there have been proposals to dedicate processor core(s) for backend network packet processing. For instance, Liu *et al.* [53] proposed a new I/O virtualization architecture called *virtualization polling engine* (VPE), where a dedicated processor core was used to execute the polling threads, which accessed the physical device and presented virtual access points to the guest VMs.

Kumar *et al.* [54] described their *Sidecore* approach that was used to reduce the interrupt virtualization overheads when using direct access network devices. They used dedicated processor cores in the hypervisor whose job was to poll the queues on the network device and signal the appropriate guest VM when a new packet was received on one of the queues. Instead, Landau *et al.* [55] proposed their *SplitX* architecture, where the hypervisor, which performed the backed packet processing, was itself run on dedicated CPU core(s). They also used limited polling to process guest VM I/O requests in the hypervisor. The main goal of both the sidecore and the splitX architectures was to reduce the number of entries and exits to and from the hypervisor. But dedicating CPU cores for packet processing could potentially lead to an under-utilization of the processor cores especially when there was no network activity. Therefore, in the Hyper-Switch architecture, the packet processing is *dynamically* offloaded to just the *idle* cores in the system.

6.6 Flow-based Packet Switching

Flow-based packet switching has recently begun to receive a lot of attention. The fundamental idea is that the packets are switched on a per-flow basis instead of a per-packet basis. This has significant advantages, as many packet processing operations can then be performed on a per-flow basis, thereby reducing the associated overheads.

The OpenFlow [63] system is perhaps the canonical example of such flow-based switching. However, the primary motivation for OpenFlow was not the efficiency of per-flow switching, but rather the flexibility afforded by removing the switching algorithm from the switch. OpenFlow was originally proposed as a way to deploy experimental network protocols over existing networks. Switches were then modified to support the OpenFlow protocol [64] that lets an external controller to program them. This allowed the controller to setup flow rules and thus, remotely manage the network traffic at the switches. These flow rules could be used to support arbitrary network protocols.

Luo *et al.* [93] proposed the use of network processors to support hardware acceleration of software OpenFlow switches. Similar to the sNICh architecture, they implemented a fast path in hardware by caching flow entries. But the focus of their work was on improving the performance of stand-alone software switches. Instead, the focus of this thesis is on last hop switching in virtualized systems. In particular, the architectures presented in this thesis optimize the inter-VM network traffic.

The flow-based interface used in the sNICh architecture has some similarities to OpenFlow. Specifically, the flow definition used in the sNICh architecture is similar to how flows are defined in OpenFlow. However, flow-based switching is used in the sNICh architecture merely to enable the decoupling of the control and data planes. In particular, the sNICh hardware does not "speak" OpenFlow. But an external controller may instead communicate with the sNICh control plane software using the OpenFlow protocol. This way OpenFlow may still be used to manage the switches in the sNICh architecture.

Open vSwitch [36] is an OpenFlow compatible software switch for commodity servers. It implements a fast in-kernel data path and a slow user-level control path. Open vSwitch is also being deployed in virtualized servers to switch VM traffic [94]. But, typically, in driver domain-based I/O architectures, both the control and data planes of Open vSwitch are implemented inside the driver domain. Instead, in the Hyper-Switch architecture, the data plane of the switch is moved from the driver domain to the hypervisor. Further, the Hyper-Switch prototype was implemented by modifying Open vSwitch. Luo *et al.* [82] proposed offloading the Open vSwitch's in-kernel data path to programmable network cards to accelerate the network traffic through Open vSwitch. In many ways, this is similar to how the data plane is implemented in the sNICh hardware. However, the sNICh architecture further leverages the proximity of the switching hardware to the server. For instance, it uses the DMA engines on the host side of the I/O bus to optimize inter-VM communication.

6.7 Miscellaneous

There have also been proposals to distribute virtual networking across all end-points within a data center [95, 96]. Here the software-based components reside on all servers that collaborate with each other and implement network virtualization and access control for VMs, while network switches are completely unaware of the individual VMs on the end-points. Virtual Distributed Ethernet (VDE) [97] is a similar tool that, based on the general concepts of LAN emulation, enables the interconnection of virtual environments via virtual Ethernet switches and virtual plugs. VMware's vSphere [98] network architecture also implements distributed virtual switches in software for uniform configuration, monitoring, and management. All these switching architectures are aimed at solving the network management problem in virtualized datacenters. While the management issues are as important as the performance issues in datacenters, it is not the direct focus of this work.

Chapter 7

Conclusions

Over the last decade, machine virtualization has been widely adopted at datacenters to reduce costs and increase efficiencies. The use of this technology has led to considerable changes in the datacenter network and the I/O subsystem within virtualized servers. In particular, the communication endpoints within the datacenter are now virtual machines (VMs), not physical servers. Consequently, the datacenter network now extends into the server.

While great strides have been made in optimizing the processor and memory virtualization solutions, the virtualization of devices, especially network devices, has lagged behind. There are significant overheads inherent in the software solutions that are commonly used to virtualize devices and perform last hop switching. Further, thanks to the increasing processor core counts on servers, server VM densities are on the rise. As a result, there is significant network traffic even between VMs hosted on a server. These trends can lead to an untenable situation in the future since the network is likely to become a performance bottleneck. This thesis presents new mechanisms and architectures to address these challenges.

First, a new mechanism for memory sharing in driver domain-based I/O architectures was proposed. The key idea in the proposed mechanism was to allow the guest VMs to unilaterally revoke the memory it had shared with the driver domain. In turn, this facilitated the reuse of shared I/O buffers by taking advantage of temporal and/or spatial locality in a guest OS's usage of I/O buffers. The new mechanism also provided a unified interface for controlled memory sharing with both driver domains and I/O devices using the new IOMMU hardware.

The proposed mechanism was evaluated using a prototype implementation in the Xen platform. In the evaluation, it reduced the CPU cost during I/O operations by up to 45% and increased the throughput by up to 150% under Xen. While the evaluation in this thesis focused on the benefits of the new mechanism for network I/O operations, it could also be used in more general scenarios. In particular, any memory sharing scenario that exhibited significant temporal and/or spatial locality should benefit from the new mechanism. For example, it should also be possible to optimize block I/O operations using the new mechanism.

Second, this thesis introduced a new software-based last hop switching architecture, called *Hyper-Switch*, for hypervisors that currently supported driver domains. The key idea in the proposed architecture was to move the last hop virtual switch from the driver domain to the hypervisor. In particular, the hypervisor implemented a highly streamlined, scalable, and efficient data plane of a flow-based software switch. So this architecture combined the best of the existing architectures. It hosted the device drivers in the driver domain to isolate any faults and the last hop virtual switch in the hypervisor to perform efficient packet switching. Further, this thesis also presented several optimizations that enabled high performance. This includes VM state-aware batching of packets to mitigate the cost of hypervisor entries and guest notifications. Preemptive copying and immediate notification of blocked guest VMs to reduce packet processing latency. Dynamic offloading of packet processing to idle CPU cores in the system to increase concurrency and make better use of the available CPU resources.

This thesis also presented a detailed evaluation of the Hyper-Switch architecture using a prototype implemented in the Xen platform. The evaluation showed that the Hyper-Switch outperformed both Xen's default network I/O architecture and KVM's vhost-net architecture. Hyper-Switch's performance was superior both in terms of the absolute bandwidth and the scalability as the number of VMs and traffic flows were varied. For instance, in the pairwise scalability experiments the Hyper-Switch achieved a peak net throughput of $\sim$81 Gbps as compared to only $\sim$31 Gbps and $\sim$47 Gbps under Xen and KVM respectively.

Third, another last hop switching architecture called *sNICh* was proposed, which was a combination of a network interface and a switching accelerator. The sNICh utilized minimal and off-the-shelf building blocks to support key elements of datacenter switching. Further, it leveraged the proximity of the switching hardware to the server by carefully dividing the network switching tasks between them. Therefore, the sNICh tool the Hyper-Switch architecture one step further by offloading the data plane of the switch to the network device and completely eliminating the need for driver domains. The sNICh architecture was evaluated using a software prototype, where the sNICh's hardware components were emulated in software. The sNICh prototype was shown to outperform and scale better than the existing solutions.

So this thesis presented a spectrum of I/O virtualization solutions that spanned both hardware and software. These solutions also illustrated the various bottlenecks and trade-offs inherent in I/O virtualization. The first two contributions of the thesis showed that it was feasible to achieve high performance without sacrificing system reliability or fault isolation when using pure software solutions. The third contribution of this thesis showed that it was feasible to implement a NIC-based solution that supports all datacenter switching functionalities.

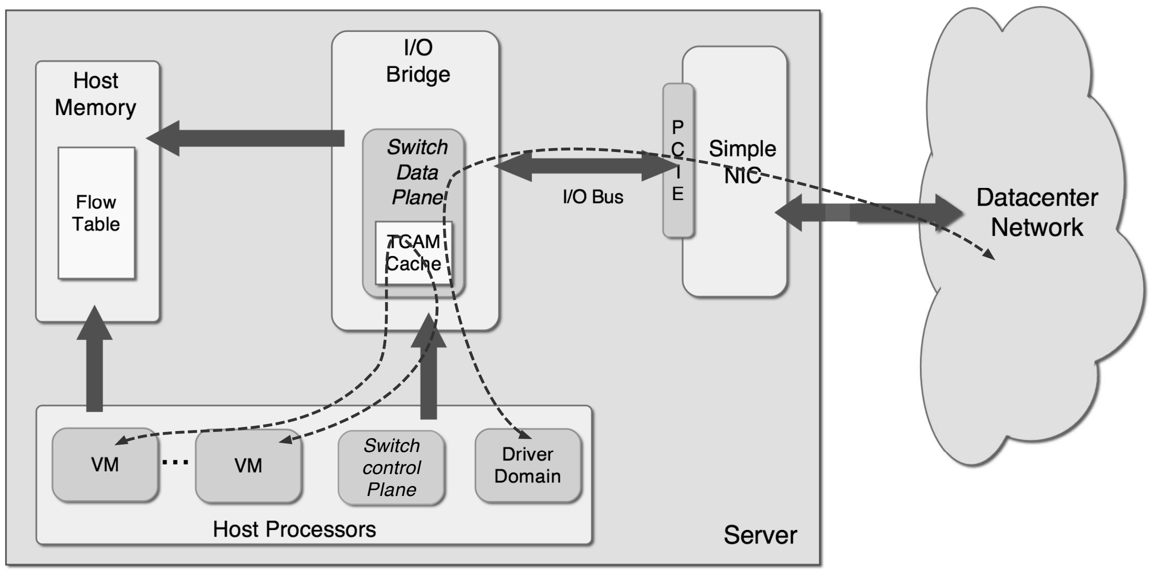


Figure 7.1 : *Future: Dedicated switching hardware on server platforms*

7.1 Future Directions

As machine virtualization gets even more widely adopted, and as networking and I/O performance become even more important, there is an urgent need to design and explore new I/O architectures. The last hop virtual switch plays a critical role in forwarding many different types of network traffic —including internal network traffic between virtual machines (VMs) co-located on the same server, external network traffic from other servers within the datacenter and clients outside the datacenter, and often, even remote storage traffic. Therefore, we strongly believe that the virtual switch implementation has to be distributed and strategically located in a virtualization platform. The architectures proposed in this thesis are a result of this philosophy. For instance, the Hyper-Switch architecture moves the virtual switch (or more precisely the data plane of the switch) from the driver domain to the hypervisor and the sNICh architecture offloads it to the network interface hardware.

These approaches that examine the blurring of the boundaries between the computing and networking infrastructure and their tighter integration, are likely to be a part of future designs. In fact, we envision the switching functionality eventually moving to dedicated chipsets within the server platforms. This idea has been previously explored in the SINIC architecture [99], where a simple network interface is integrated on the processor.

Figure 7.1 illustrates the idea of having dedicated switching hardware on server platforms. This architecture is based on the sNICh architecture presented in this thesis. It should also be possible to continue hosting the device drivers in their own driver domains. Then, external packets would still be forwarded through these driver domains and the switching hardware does not need to directly communicate with the network interface(s). This is similar to how external network traffic is handled in the Hyper-Switch architecture. Further, as an optimization, the limitations due to the TCAM size restrictions on the flow table cache can be alleviated using a larger in-memory flow table. The switch can use the TCAM-based flow table cache in hardware, but upon a cache miss, the packet does not need to be shipped up to the software. Instead, the hardware can access a larger flow table directly in main memory. In many ways this is analogous to how the page table is handled for virtual address translation. The small, fast TCAM lookup table is analogous to the TLB. When there is no match in the TCAM (analogous to a TLB miss), the hardware can directly traverse a larger structure and potentially install a new flow rule into the TCAM (analogous to a hardware page table walk).

The various hardware/software architectures presented in thesis pave the way for this vision of a distributed solution where the data plane of the last hop virtual switch is implemented on a dedicated hardware within the server platform. Such approaches are going to be critical in ensuring that networking is not performance bottleneck in virtualized many-core systems in future. The solutions presented in this thesis is a step in that direction.

Bibliography

[1] W. Vogels, “Beyond server consolidation,” *ACM Queue*, vol. 6, pp. 20–26, January/February 2008.

[2] Amazon Web Services, “Amazon elastic compute cloud (EC2).” `http://aws.amazon.com/ec2/`.

[3] Rackspace, US Inc., “Rackspace cloud.” `http://www.rackspace.com/cloud/`.

[4] Microsoft, “Windows Azure.” `http://www.windowsazure.com/en-us/`.

[5] IDC, “Worldwide market for enterprise server virtualization to reach $19.3 billion by 2014, according to IDC.” `http://www.idc.com/about/viewpressrelease.jsp?containerId=prUS22605110`, December 2010.

[6] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, “The nature of data center traffic: Measurements & analysis,” in *IMC ’09: Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, pp. 202–208, November 2009.

[7] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, “The cost of a cloud: Research problems in data center networks,” *SIGCOMM Computer Communication Review*, vol. 39, pp. 68–73, January 2009.

[8] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williams, “Safe hardware access with the Xen virtual machine monitor,” in *OASIS ’04:*

Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT Infrastructure, October 2004.

[9] Microsoft, "Hyper-V architecture." `http://msdn.microsoft.com/en-us/library/cc768520.aspx`.

[10] S. Devine, E. Bugnion, and M. Rosenblum, "Virtualization system including a virtual machine monitor for a computer with a segmented architecture," *US Patent #6,397,242*, October 1998.

[11] Qumranet, "KVM: Kernel-based virtualization driver." `http://www.redhat.com/f/pdf/rhev/DOC-KVM.pdf`, 2009.

[12] J. Liu, W. Huang, B. Abali, and D. K. Panda, "High performance VMM-bypass I/O in virtual machines," in *ATC '06: Proceedings of the USENIX Annual Technical Conference*, June 2006.

[13] 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 *HPCA '07: Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, February 2007.

[14] H. Raj and K. Schwan, "High performance and scalable I/O virtualization via self-virtualized devices," in *HPDC '07: Proceedings of the IEEE International Symposium on High Performance Distributed Computing*, June 2007.

[15] K. Mansley, G. Law, D. Riddoch, G. Barzini, N. Turton, and S. Pope, "Getting 10 Gb/s from Xen: Safe and fast device access from unprivileged domains," in *Proceedings of the Euro-Par Workshop on Parallel Processing*, pp. 224–233, August 2007.

[16] S. Rixner, "Network virtualization: Breaking the performance barrier," *ACM Queue*, vol. 6, pp. 36–52, January/February 2008.

[17] VMware, Inc., "VMware virtual networking concepts." `http://www.vmware.com/files/pdf/virtual_networking_concepts.pdf`, 2007.

[18] "Linux Ethernet bridge." `http://www.linuxfoundation.org/collaborate/workgroups/networking/bridge`, November 2009.

[19] Cisco Systems, Inc., "Cisco Nexus 1000V series switches." `http://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps9902/data_sheet_c78-492971.pdf`, August 2011.

[20] P. Congdon, "Virtual Ethernet port aggregator." `http://www.ieee802.org/1/files/public/docs2008/new-congdon-vepa-1108-v01.pdf`, November 2008.

[21] J. Pelissier, "VNTag 101." `http://www.ieee802.org/1/files/public/docs2009/new-pelissier-vntag-seminar-0508.pdf`, 2009.

[22] J. M. Kaplan, W. Forrest, and N. Kindler, "Revolutionizing data center energy efficiency," tech. rep., McKinsey and Company, July 2008. `http://www.mckinsey.com/clientservice/bto/pointofview/pdf/Revolutionizing_Data_Center_Efficiency.pdf`.

[23] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS Operating Systems Review*, vol. 41, pp. 205–220, December 2007.

[24] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pp. 164–177, October 2003.

[25] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz, "Unmodified device driver reuse and improved system dependability via virtual machines," in *OSDI '04: Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pp. 17–30, December 2004.

[26] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler, "An empirical study of operating system errors," in *SOSP '01: Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pp. 73–88, October 2001.

[27] PCI-SIG, "Single Root I/O Virtualization." `http://www.pcisig.com/specifications/iov/single_root`.

[28] Intel Corporation, "Intel 82599 10 GbE controller datasheet." `http://download.intel.com/design/network/datashts/82599_datasheet.pdf`, October 2011. Revision 2.72.

[29] Chelsio Communications, "The Chelsio Terminator 4 ASIC." `http://chelsio.com/assetlibrary/whitepapers/Chelsio%20T4%20Architecture%20White%20Paper.pdf`, 2010.

[30] Broadcom Corporation, "BCM57712 product brief." `http://www.broadcom.com/collateral/pb/57712-PB00-R.pdf`, January 2010.

[31] M. Nelson, B.-H. Lim, and G. Hutchins, "Fast transparent migration for virtual

machines," in *ATC '05: Proceedings of the USENIX Annual Technical Conference*, April 2005.

[32] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. Lam, and M. Rosenblum, "Optimizing the migration of virtual computers," in *OSDI '02: Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pp. 377–390, December 2002.

[33] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *NSDI '05: Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation*, pp. 273–286, May 2005.

[34] K. K. Ram, J. R. Santos, Y. Turner, A. L. Cox, and S. Rixner, "Achieving 10 Gb/s using safe and transparent network interface virtualization," in *VEE '09: Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pp. 61–70, March 2009.

[35] J. Wiegert, G. Regnier, and J. Jackson, "Challenges for scalable networking in a virtualized server," in *ICCCN '07: Proceedings of the 16th International Conference on Computer Communications and Networks*, pp. 179–184, August 2007.

[36] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker, "Extending networking into the virtualization layer," in *HotNets-VIII: Proceedings of the Workshop on Hot Topics in Networks*, October 2009.

[37] S. Chinni and R. Hiremane, "Virtual machine device queues." `http://software.intel.com/file/1919`, 2007.

[38] 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, August 2006.

[39] Advanced Micro Devices, Inc., "AMD I/O virtualization technology (IOMMU) specification." `http://support.amd.com/us/Processor_TechDocs/34434-IOMMU-Rev_1.26_2-11-09.pdf`, February 2007.

[40] M. Ben-Yehuda, J. Mason, O. Krieger, J. Xenidis, L. Van Doorn, A. Mallick, J. Nakajima, and E. Wahlig, "Utilizing IOMMUs for virtualization in Linux and Xen," in *OLS '06: Proceedings of the Ottawa Linux Symposium*, July 2006.

[41] J. R. Santos, Y. Turner, G. Janakiraman, and I. Pratt, "Bridging the gap between software and hardware techniques for I/O virtualization," in *ATC '08: Proceedings of the USENIX Annual Technical Conference*, pp. 29–42, June 2008.

[42] M. Ben-Yehuda, J. Xenidis, M. Ostrowski, K. Rister, A. Bruemmer, and L. Van Doorn, "The price of safety: Evaluating IOMMU performance," in *OLS '07: Proceedings of the Ottawa Linux Symposium*, June 2007.

[43] P. Willmann, S. Rixner, and A. L. Cox, "Protection strategies for direct access to virtualized I/O devices," in *ATC '08: Proceedings of the USENIX Annual Technical Conference*, June 2008.

[44] N. Amit, M. Ben-Yehuda, D. Tsafrir, and A. Schuster, "vIOMMU: Efficient IOMMU emulation," in *ATC '11: Proceedings of the USENIX Annual Technical Conference*, June 2011.

[45] "Netperf: A network performance benchmark." `http://www.netperf.org`, 1995. Revision 2.5.

[46] J. Levon, "OProfile, a system-wide profiler for Linux systems." `http://oprofile.sourceforge.net`.

[47] A. Menon, J. R. Santos, Y. Turner, G. Janakiraman, and W. Zwaenepoel, "Diagnosing performance overheads in the Xen virtual machine environment," in *VEE '05: Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, pp. 13–23, June 2005.

[48] D. Ongaro, A. L. Cox, and S. Rixner, "Scheduling I/O in virtual machine monitors," in *VEE '08: Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, March 2008.

[49] K. K. Ram, J. R. Santos, and Y. Turner, "Redesigning Xen's memory sharing mechanism for safe and efficient I/O virtualization," in *WIOV '10: Proceedings of the 3rd Workshop on I/O Virtualization*, March 2010.

[50] S. Govindan, A. R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam, "Xen and co.: Communication-aware CPU scheduling for consolidated Xen-based hosting platforms," in *VEE '07: Proceedings of the 3rd International Conference on Virtual Execution Environments*, pp. 126–136, June 2007.

[51] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee, "Task-aware virtual machine scheduling for I/O performance," in *VEE '09: Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pp. 101–110, March 2009.

[52] G. Liao, D. Guo, L. Bhuyan, and S. R. King, "Software techniques to improve virtualized I/O performance on multi-core systems," in *ANCS '08: Proceedings*

of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, pp. 161–170, November 2008.

[53] J. Liu and B. Abali, "Virtualization Polling Engine (VPE): Using dedicated CPU cores to accelerate I/O virtualization," in *ICS '09: Proceedings of the ACM/IEEE International Conference on Super Computing*, pp. 225–234, November 2009.

[54] S. Kumar, H. Raj, K. Schwan, and I. Ganev, "Re-architecting VMMs for multicore systems: The sidecore approach," in *WIOSCA '07: Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture*, June 2007.

[55] A. Landau, M. Ben-Yehuda, and A. Gordon, "SplitX: Split guest/hypervisor execution on multi-core," in *WIOV '11: Proceedings of the 4th Workshop on I/O Virtualization*, May 2011.

[56] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat, "Enforcing performance isolation across virtual machines in Xen," in *Proceedings of the 7th ACM/IFIP/USENIX Middleware Conference*, November 2006.

[57] A. Porterfield, R. Fowler, A. Mandal, and M. Y. Lim, "Empirical evaluation of multi-core memory concurrency," tech. rep., RENCI, January 2009. `www.renci.org/publications/techreports/TR-09-01.pdf`.

[58] Advanced Micro Devices, Inc., "Shared level-1 instruction-cache performance on AMD family 15h CPUs," December 2011.

[59] L. Zhao, L. N. Bhuyan, R. Iyer, S. Makineni, and D. Newell, "Hardware support for accelerating data movement in server platform," *IEEE Transactions on*

Computers, vol. 56, pp. 740–753, January 2007.

[60] K. K. Ram, J. R. Santos, Y. Turner, A. L. Cox, and S. Rixner, "Redesigning Xen memory sharing (grant) mechanism," in *Xen Summit North America*, August 2011.

[61] Cisco Systems, Inc., "Cisco Nexus 5000 series architecture: The building blocks of the unified fabric." `http://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps9670/white_paper_c11-462176.pdf`, June 2009.

[62] HP Procurve Networking, "ACL manual for 3500-8200 series of switches." `http://cdn.procurve.com/training/Manuals/3500-5400-6200-8200-ASG-Jan08-10-ACLs.pdf`, January 2008.

[63] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM Computer Communication Review*, vol. 38, pp. 69–74, April 2008.

[64] "Openflow switch specification (version 1.0.0)." `http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf`, February 2011. Version 1.1.0.

[65] H. Song, J. Turner, and S. Dharmapurikar, "Packet classification using coarse-grained tuple spaces," in *ANCS '06: Proceedings of the 2nd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, pp. 41–50, December 2006.

[66] Intel Corporation, "Accelerating high-speed networking with Intel I/O Acceleration Technology." `http://download.intel.com/technology/comms/perfnet/download/98856.pdf`, April 2007.

[67] "Netfilter: Firewalling, NAT, and packet mangling for Linux." `http://www.netfilter.org`.

[68] D. E. Taylor and J. S. Turner, "ClassBench: A packet classification benchmark," in *INFOCOM '05: Proceedings of the Annual Joint Conference of the IEEE Computer and Communications Societies*, pp. 2068–2079, March 2005.

[69] J. Sugerman, G. Venkitachalam, and B.-H. Lim, "Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor," in *ATC '01: Proceedings of the USENIX Annual Technical Conference*, pp. 1–14, June 2001.

[70] A. Whitaker, M. Shaw, and S. D. Gribble, "Scale and performance in the Denali isolation kernel," in *OSDI '02: Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, December 2002.

[71] R. Russell, "virtio: Towards a de-facto standard for virtual I/O devices," *SIGOPS Operating Systems Review*, vol. 42, pp. 95–103, July 2008.

[72] A. Menon, A. L. Cox, and W. Zwaenepoel, "Optimizing network virtualization in Xen," in *ATC '06: Proceedings of the USENIX Annual Technical Conference*, June 2006.

[73] R. Huggahalli, R. Iyer, and S. Tetrick, "Direct cache access for high bandwidth network I/O," in *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pp. 50–59, IEEE Computer Society, June 2005.

[74] Intel Corporation, "Intel 82598 10 GbE Ethernet controller open source datasheet." `http://download.intel.com/design/network/datashts/319282.pdf`, December 2010. Revision 3.21.

[75] Intel Corporation, "PCI-SIG SR-IOV primer." `http://www.intel.com/content/dam/doc/application-note/pci-sig-sr-iov-primer-sr-iov-technology-paper.pdf`, January 2011. Revision 2.5.

[76] X. Zhang, S. McIntosh, P. Rohatgi, and J. L. Griffin, "Xensocket: A high-throughput transport for virtual machines," in *Middleware '07: Proceedings of the 8th International Middleware Conference*, pp. 184–203, November 2007.

[77] K. Kim, C. Kim, S.-I. Jung, H.-S. Shin, and J.-S. Kim, "Inter-domain socket communications supporting high performance and full binary compatibility on Xen," in *VEE '08: Proceedings of the 4th SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pp. 11–20, March 2008.

[78] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia, "RFC 5040: A remote direct memory access protocol specification," October 2007.

[79] L. Rizzo and G. Lettieri, "VALE, a switched Ethernet for virtual machines." `http://info.iet.unipi.it/~luigi/%papers/20120608-vale.pdf`, June 2012.

[80] L. Rizzo, "Netmap: a novel framework for fast packet I/O," in *ATC '12: Proceedings of the USENIX Annual Technical Conference*, June 2012.

[81] VMware, Inc., "VMware vSphere 4.1 networking performance." `http://www.vmware.com/files/pdf/techpaper/Performance-Networking-vSphere4-1-WP.pdf`, April 2011.

[82] Y. Luo, E. Murray, and T. Ficarra, "Accelerated virtual switching with programmable NICs for scalable data center networking," in *VISA '10: Proceedings*

of the 2nd ACM SIGCOMM Workshop on Virtualized Infrastructure Systems and Architectures, September 2010.

[83] S. Makineni and R. Iyer, "Performance characterization of TCP/IP packet processing in commercial server workloads," in *WWC-6 '03: Proceedings of the IEEE 6th Annual Workshop on Workload Characterization*, pp. 33–41, October 2003.

[84] J. C. Mogul, "TCP offload is a dumb idea whose time has come," in *HOTOS'03: Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, pp. 25–30, May 2003.

[85] H. Kim and S. Rixner, "TCP offload through connection handoff," *SIGOPS Operating Systems Review*, vol. 40, pp. 279–290, April 2006.

[86] M. S. Schlansker, N. Chitlur, E. Oertli, P. M. Stillwell Jr., L. Rankin, D. Bradford, R. J. Carter, J. Mudigonda, N. L. Binkert, and N. P. Jouppi, "High-performance Ethernet-based communications for future multi-core processors," in *ICS '07: Proceedings of the ACM/IEEE International Conference on Super Computing*, pp. 1–12, November 2007.

[87] Microsoft Corporation, "Scalable networking with RSS." `http://www.microsoft.com/whdc/device/network/NDIS_RSS.mspx`, November 2008.

[88] C. Eddington, "Infinibridge: An Infiniband channel adapter with integrated switch," *IEEE Micro*, vol. 22, pp. 48–56, March/April 2002.

[89] J. Shafer and S. Rixner, "RiceNIC: A reconfigurable network interface for experimental research and education," in *ExpCS '07: Proceedings of the workshop on Experimental Computer Science*, June 2007.

[90] J. Naous, G. Gibb, S. Bolouki, and N. McKeown, "NetFPGA: Reusable router architecture for experimental research," in *PRESTO '08: Proceedings of the ACM workshop on Programmable Routers for Extensible Services of TOmorrow*, pp. 1–7, August 2008.

[91] J. Naous, D. Erickson, A. G. Covington, G. Appenzeller, and N. McKeown, "Implementing an OpenFlow switch on the NetFPGA platform," in *ANCS '08: Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pp. 1–9, November 2008.

[92] G. Regnier, S. Makineni, R. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong, "TCP onloading for data center servers," *IEEE Computer*, vol. 37, pp. 48–58, November 2004.

[93] Y. Luo, P. Cascon, E. Murray, and J. O. Lopera, "Accelerating OpenFlow switching with network processors," in *ANCS '09 (Poster)*, October 2009.

[94] J. Pettit, J. Gross, B. Pfaff, M. Casado, and S. Crosby, "Virtual switching in an era of advanced edges," in *DC-CAVES '10: Proceedings of the 2nd Workshop on Data Center Converged and Virtual Ethernet Switching*, September 2010.

[95] S. Cabuk, C. I. Dalton, H. Ramasamy, and M. Schunter, "Towards automated provisioning of secure virtualized networks," in *CCS '08: Proceedings of the 15th ACM Conference on Computer and Communications Security*, pp. 235–245, October 2007.

[96] A. Edwards, A. Fischer, and A. Lain, "Diverter: A new approach to networking within virtualized infrastructures," in *WREN '09: Proceedings of the ACM SIGCOMM Workshop: Research on Enterprise Networking*, August 2009.

[97] R. Davoli, "VDE: Virtual distributed Ethernet," *TRIDENTCOM '05: Proceedings of the 1st International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities*, pp. 213–220, February 2005.

[98] VMware, Inc., "What's new in VMware vSphere 4: Virtual networking." `http://www.vmware.com/files/pdf/VMW_09Q1_WP_vSphereNetworking_P8_R1.pdf`, 2009.

[99] N. L. Binkert, A. G. Saidi, and S. K. Reinhardt, "Integrated network interfaces for high-bandwidth TCP/IP," in *ASPLOS '06: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006.