Hyper-Switch: A Scalable Software Virtual Switching Architecture

Kaushik Kumar Ram, Alan L. Cox, Mehul Chadha and Scott Rixner

Rice University

Abstract

In virtualized datacenters, the last hop switching happens inside a server. As the number of virtual machines hosted on the server goes up, the last hop switch can be a performance bottleneck. This paper presents the *Hyper-Switch*, a highly efficient and scalable software-based network switch for virtualization platforms that support driver domains. It combines the best of the existing I/O virtualization architectures by hosting device drivers in a driver domain to isolate faults and placing the packet switch in the hypervisor for efficiency. In addition, this paper presents several optimizations that enhance performance. They include virtual machine (VM) state-aware batching of packets to mitigate the costs of hypervisor entries and guest notifications, preemptive copying and immediate notification of blocked VMs to reduce packet arrival latency, and, whenever possible, packet processing is dynamically offloaded to idle processor cores. These optimizations enable efficient packet processing, better utilization of the available CPU resources, and higher concurrency.

We implemented a Hyper-Switch prototype in the Xen virtualization platform. This prototype's performance was then compared to Xen's default network I/O architecture and KVM's vhost-net architecture. The Hyper-Switch prototype performed better than both, especially for inter-VM network communication. For instance, in one scalability experiment measuring aggregate inter-VM network throughput, the Hyper-Switch achieved a peak of ∼81 Gbps as compared to only ∼31 Gbps under Xen and ∼47 Gbps under KVM.

1 Introduction

Machine virtualization is now used extensively in datacenters. This has led to considerable change to both 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.

At the same time, thanks to increasing core counts on processors, server VM densities is 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—especially between VMs—in virtualized servers.

There are many I/O architectures for network communication in virtualized systems. Of these, software device virtualization is the most widely used. This preference for software over specialized hardware devices is due in part to the rich set of features—including security, isolation, and mobility—that the software solutions offer.

The software solutions can be further divided into *driver domain* and *hypervisor-based* architectures. Driver domains are dedicated VMs that host the drivers that are used to access the physical devices. It provides a safe execution environment for the device drivers. Arguably, the hypervisors that support driver domains are more robust and fault tolerant, as compared to 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 [29, 31].

In existing I/O architectures, the virtual switch is implemented inside the same software domain where the virtual devices are implemented and the device drivers are hosted. For instance, all of these components are implemented inside a driver domain in Xen [13] and the hypervisor in KVM [26]. This collocation is purely a matter of convenience since packets must be switched when they are moved between the virtual devices and the device drivers.

In this paper, we introduce the *Hyper-Switch*, which challenges the existing convention by separating the virtual switch from the domain that hosts the device drivers. The Hyper-Switch is a highly efficient and scalable software-based switch for virtualization platforms 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 small relative to the size of the switch control plane, it does not significantly increase the size of the hypervisor or the platform's overall trusted computing base (TCB). The *Hyper-Switch* explores a new point in the virtual switching design space.

Another contribution of this paper is a set of optimizations that increase performance. They enable the Hyper-Switch to efficiently support both bulk and latency sensitive network traffic. They include *VM state-aware batching* of packets to mitigate the costs of hypervisor entries on the transmit side and guest notifications on the receive side. *Preemptive copying* is employed at the receiving VM, when it is being notified of packet arrival, to reduce latency. Further, whenever possible, packet processing is *dynamically offloaded* to idle processor cores. The offloading is performed using a low-overhead mechanism that takes into account 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 data plane's integration within the hypervisor and its proximity to the scheduler. As a result, the Hyper-Switch enables much improved and scalable network performance, while maintaining the robustness and fault tolerance that derive from the use of 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.

We evaluated the Hyper-Switch using a prototype that was implemented in the Xen virtualization platform [4]. The prototype was built by modifying Open vSwitch [24], a multi-layer software switch for commodity servers. In this evaluation, the Hyper-Switch's performance was compared to that of KVM's vhost-net and Xen's default network I/O architectures. The results show that the Hyper-Switch enables much better scalability and peak throughput than both of these existing architectures.

The rest of this paper is organized as follows. Section 2 further motivates the Hyper-Switch by discussing some of the issues with existing architectures. Section 3 explains the Hyper-Switch's design. Section 4 describes the implementation of the Hyper-Switch prototype. Section 5 presents a detailed evaluation of the Hyper-Switch. Section 6 discusses related work. Finally, Section 7 summarizes the conclusions.

2 Motivation

The need for efficient and scalable network communication within virtualized servers is increasing. Intel already claims to have an architecture that can scale to 1000 cores on a single chip [15]. Furthermore, the number of cores on a chip is predicted to grow to 64 in a few years and to 256–512 by the end of the decade [12]. If this last prediction is borne out, then in 2020 a single 1U server will have as many cores as an entire rack of servers does today.

In addition, communication between servers within the same datacenter already accounts for a significant fraction of the datacenter's total network traffic [14]. Moreover, Benson *et al*.'s study of multiple datacenter networks reported that 80% of the traffic originating at servers in cloud datacenters never leaves the rack [5]. If the predictions for the growing number of cores come to pass, then a rack of servers may be replaced by VMs within a single physical server, and the network traffic that today never leaves the rack may instead never leave the server. Consequently, the Hyper-Switch has been heavily optimized to enable high performance inter-VM communication.

Modern multi-core systems enable concurrent processing of network packets. Under Xen's default network architecture, the driver domain can run in parallel to the transmitting and receiving VMs. Consequently, it is possible to perform packet switching in parallel with packet transmission and reception. However, there are several fundamental problems with traditional driver domain architectures that limit I/O performance scalability. Fundamentally, the driver domain must be scheduled to run whenever packets are waiting to be processed. This might involve scheduling multiple virtual processors depending on the number of threads used for packet processing in the driver domain. As a result, 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.

Today, it is standard practice in real-world virtualization deployments to dedicate processor cores to the driver domain. This avoids scheduling delays, but often leaves cores idle. 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 cores—including Kumar *et al*.'s sidecore approach [17], and Landau *et al*.'s split execution (SplitX) model [18]. But, this can lead to underutilization of these cores. Further, this goes against one of the fundamental tenets of virtualization, which is to enable the most efficient utilization of the server resources. Hence the Hyper-Switch has been designed to smartly and dynamically utilize the available resources.

At the same time, reliability cannot be ignored, especially as servers in datacenters move toward multi-tenancy. Hypervisors that support driver domains are potentially more robust and fault tolerant. However, driver domains incur significant overheads. These overheads are due to the costs of moving packets between the guest VMs and the driver domain [21, 29, 31], because the driver domain cannot trivially access a packet in the guest VM's memory. The driver domain is just

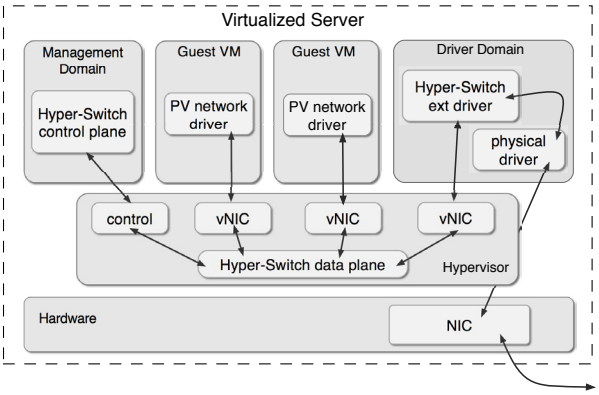

Figure 1: *The Hyper-switch architecture. The last hop virtual switch is implemented partly in the hypervisor (data plane) and partly in the management layer (control plane). The device drivers are hosted in the driver domain.*

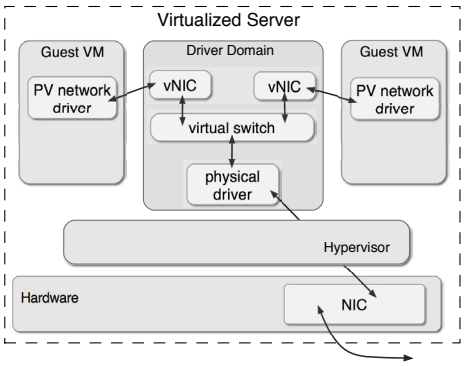

Figure 2: *Traditional driver domain architecture. The driver domain hosts the device drivers and the last hop virtual switch.*

another VM and the hypervisor maintains memory isolation between VMs. So, the driver domain must use an expensive memory sharing mechanism provided to access the packet. Hypervisor-based architectures do not incur these memory sharing overheads since the packets in the guest VMs' memory can be directly accessed from the hypervisor. The Hyper-Switch has been designed to take advantage of driver domains without incurring the associated memory sharing overheads.

3 Hyper-Switch Design

Figure 1 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 collocated on the same server is handled entirely within 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 VM through the Hyper-Switch. For outgoing external traffic, these two steps are simply reversed. 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. Contrast this with the traditional driver domain architecture as illustrated in Figure 2.

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 Open vSwitch [24]. However, Open vSwitch's control and data planes are both implemented inside the driver domain.

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

3.1 Basic Design

Packet processing by the Hyper-Switch begins at the transmitting VM (or driver domain) where the packet originates and ends at the receiving VM (or driver domain) where the packet has to be delivered. It proceeds in four stages: (1) packet transmission, (2) packet switching, (3) packet copying, and (4) packet reception.

Packet Transmission. In the first stage, the transmitting VM pushes the packet to the Hyper-Switch for processing. Packet transmission begins when the guest VM's network stack forwards the packet to its paravirtualized network driver. Then the packet is queued for transmission by setting up descriptors in the transmit ring. A single packet can potentially span multiple descriptors depending on its size. Typically, packets are never segmented in the transmitting guest VM. So the packets belonging to internal network traffic can be forwarded as is. The external packets are segmented either in the driver domain or the network hardware. Today, segmentation offload is a standard feature in most network devices.

Packet Switching. In the second stage of packet processing, the packet is switched to determine its destination. Switching is triggered by a hypercall from the

transmitting VM. It begins with reading the transmit ring to find new packets. Each packet is then pushed to the Hyper-Switch's data plane where it is processed using flow-based packet switching. The data plane must be able to read the packet's headers in order to switch it. Since the data plane is located in the hypervisor, which has direct access to every VM's memory, it can read the headers directly from the transmitting VM's memory.

Packet switching by the data plane proceeds as follows: (1) The packet header fields are parsed to identify the corresponding packet flow. (2) The packet flow is used to lookup a matching flow rule in a software flow table. When this lookup fails, the packet is forwarded to the control plane in the management layer. (3) A successful flow table lookup 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 destination ports or to drop the packet. Each port has an internal receive queue where the switched packet is temporarily placed. This port corresponds to a virtual network interface (vNIC) within the destination 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 hypervisor'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 hypervisor's data plane.

Packet Copying. In the third stage of packet processing, the switched packet is copied into the receiving VM's memory. Empty buffers for receiving new packets are provided through the vNIC. Specifically, the descriptors in the receive ring provide the address of the empty buffers in the VM's memory.

By default, the destination VM is responsible for performing packet copies. Once switching is completed, the destination VM is notified via a virtual interrupt. Subsequently, that VM issues a hypercall. While in the hypervisor, it dequeues the packet from the port's internal receive queue, and copies the packet into the memory given by the next descriptor in the receive ring. The packet is copied directly from the transmitting VM's memory to the receiving VM's memory. After which, the memory that was allocated for this packet at various places—inside the hypervisor and in the transmitting VM—is released.

Packet Reception. In the fourth and final stage, the para-virtualized network driver in the receiving 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. However, under the Hyper-Switch, the receiving VM was already notified in the previous stage. So packet reception can happen as soon as the hypercall for copying the packet is complete. The new packet is then pushed into the receiving VM's network stack.

3.2 Preemptive Packet Copying

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 calls into the hypervisor to copy the packet. However, delivering a notification to a VM already requires entry into the hypervisor. Under Xen, when there is a pending notification to a VM, the VM is interrupted and pulled inside the hypervisor. Since hypercalls are expensive operations, the Hyper-Switch tries to avoid them. In this case, it takes advantage of the hypervisor entry upon event notification to avoid a separate hypercall to perform the packet copy. Instead, the packet copy is performed *preemptively* when the receiving VM is being notified. In essence, the packet copy operation is combined with the notification to the receiving VM. This optimization avoids one hypervisor entry for every packet that is delivered to a VM.

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 before, even this notification requires hypervisor intervention.[1] Therefore, despite the preemptive packet copy optimization, the overhead of entering the hypervisor is incurred multiple times on every packet.

To mitigate this overhead, we use *VM state-aware batching*, 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

[1] 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 again enters the hypervisor to return from the exception context.

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.

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 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 can be achieved by polling: (1) all the internal receive queues, looking for packets waiting to be copied, and (2) all the transmit rings, looking for packets waiting to be switched. This can be performed by processor cores that are currently idle. 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.

Instead, 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 vNIC. Then one of the idle cores is chosen and woken up to perform the packet copy. A low-overhead mechanism is used to offload work to the idle cores. It uses a simple interprocessor messaging facility to request a specific idle core to copy packets at a specific vNIC. Further, this mechanism attempts to spread the work across many idle cores. Otherwise, if all the work is offloaded to a single idle core, it might become a 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, offloading packet switching to idle cores is not trivial. In the common case, packets are asynchronously queued by the transmitting VM without entering the hypervisor. So it is not possible to offload the switching tasks precisely when packets are queued. 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 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:[2] (1) The transmitting VM, (2) the packet copier, and (3) the receiving VM. So the packet data can be potentially brought into three different CPU caches depending on the system's cache hierarchy and where the two VMs and the packet 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. But if the packet copier runs on an idle core, the access latency will depend on whether the idle core shares any cache with the receiving VM's core. Therefore, under Hyper-Switch, the offload mechanism for packet processing is optimized to take advantage of 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. 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 interprocessor 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.

3.5 More Packet Processing Opportunities

A packet that is queued in the transmit ring at a vNIC will *eventually* be switched by either the transmitting VM or

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

an idle core. This might happen immediately if an idle core polls this interface looking for packets waiting to be switched or it might happen only when the transmit timer period that is implemented by VM-state aware batching elapses.[3]

Consider a VM that queues some packets for transmission at its vNIC and then blocks. Let's assume that there are no other idle cores. 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 core becomes idle after the VM blocks, there is no guarantee that the blocked 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 VM can block despite its packets waiting to be switched.

When a VM's virtual processor blocks, it has to enter the hypervisor to give up its core. 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 VM. Consequently, instead of blocking, the VM returns to process the packets that were just received.

4 Implementation Details

We implemented a prototype of the Hyper-Switch architecture, which is depicted in Figure 3. We implemented the switch's data plane by porting parts of Open vSwitch to the Xen hypervisor. Open vSwitch's control plane was used without modification. We also developed a new para-virtualized (PV) network interface 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.

Open vSwitch Overview. Open vSwitch [24] is an OpenFlow compatible, multi-layer software switch for commodity servers. The control and data planes are separated. 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. 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). In the common case, the network traffic between the guest VMs is directly switched by Open vSwitch's data plane within the kernel. Open vSwitch provides a *vport* abstraction that can be bound to any network interface in the driver domain. In addition, there is one vport for every vNIC in the system.

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

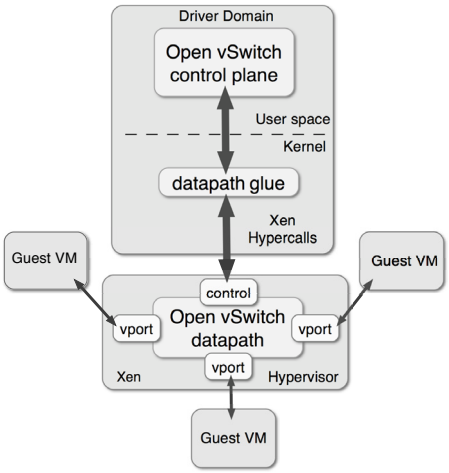

Figure 3: *Hyper-Switch prototype. It was built by porting essential parts of Open vSwitch's datapath to the Xen hypervisor.*

Porting Open vSwitch's Datapath. We implemented the Hyper-Switch's data plane by porting Open vSwitch's datapath to the Xen hypervisor. 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.

The driver domain kernel also included a datapath glue layer to enable communication between the control and data planes. This layer converted the commands from Open vSwitch's control plane 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.

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 fixed circular buffers where the producer and consumer(s) could access the ring descriptors concurrently. The interface also included an internal receive queue that contained packets that were yet to be copied into the receiving VM's memory.

Hypervisor Integration. As explained in Section 3.2, packet copying was preemptively performed by combining it with the notification to the receiving VM. We implemented this by checking for packets to copy when the associated virtual interrupt was delivered by the Xen hypervisor to a VM. Further, packet switching and copying were also performed when a VM *voluntarily* blocked.

Thus the VM's vNIC was polled for packets to be copied or switched, just before the scheduler was invoked to yield the processor and find another VM to run.

Offloading Packet Processing. We implemented the offloading of packet processing 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 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. Finally, when there is no more work to be done, it enters one of the sleep states to save power.

In the Hyper-Switch architecture, we extended Xen's idle loop to copy and switch packets. 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 were chosen to select an idle core that made the best use of the CPU caches. 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 interprocessor 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. 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 higher priority work to be done, then the offloaded packet processing was aborted.

5 Evaluation

This section presents a detailed evaluation of the Hyper-Switch architecture. The evaluation was performed using the Hyper-Switch prototype in Xen. The primary goal of this evaluation was to compare Hyper-Switch with existing architectures that implement the virtual switch either entirely within the driver domain or entirely within the hypervisor. Toward this end, the end-to-end performance under Hyper-Switch was compared to that under Xen's default driver domain-based architecture and KVM's hypervisor-based architecture.

5.1 Experimental Setup and Methodology

The experiments were run on a 32-core server with two 2.2 GHz AMD Opteron 6274 processors and 64 GB of memory. This processor is based on AMD's Bulldozer micro-architecture where two cores (called a *module*) share the second level data cache (L2) and the instruction caches (L1i and L2i). Further, four modules (called a *node*) share the unified third level cache (L3). And each Opteron 6274 processor includes two such nodes. Under Xen,[4] the server was configured to run up to 32 para-virtualized (PV) Linux guest VMs (v2.6.38 pvops) and one PV Linux driver domain (v2.6.38 pvops), in addition to the privileged management domain 0 (Linux v3.4.4 pvops). The PV linux guests use a specialized network driver which is optimized for the virtual network interface that the hypervisor provides to the VMs. 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. The server was directly connected to an external client using a 10 Gbps Ethernet link. The client consisted of 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. The CPUs at the external client were never a performance bottleneck in any of the experiments.

The netperf microbenchmark [2] was used in all the experiments to generate network traffic. In particular, netperf was used to create two types of network traffic: (1) TCP stream and (2) UDP request/response traffic. The TCP stream traffic was used to measure the achievable throughput. The UDP request/response traffic was used to measure the packet processing latency. Unless otherwise specified, the `sendfile` option was used on the transmit side in all experiments. The performance of Hyper-Switch was compared to the performance of Open vSwitch under both Xen [13] and KVM [26]. Para-virtualized network interfaces were used in all these systems. In the rest of this section, we use "KVM" to refer to the performance of Open vSwitch under KVM. Similarly, we use "Xen" to refer to the performance of Open vSwitch under Xen's default network I/O architecture. This should not be confused with the Hyper-Switch prototype that is also implemented in Xen.

Open vSwitch under Xen. In Xen, Open vSwitch is implemented entirely in the driver domain. Under Xen, 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. Netback is multi-threaded, and there is one netback (kernel) thread for every vCPU in the driver domain. Each guest VM's vNIC is bound to one of these threads. The packets associated with a specific vNIC are processed only by the thread to which it is bound. The recom-

[4] Xen v4.2 - mainline git repository (`xen-unstable.git`) May 2012.

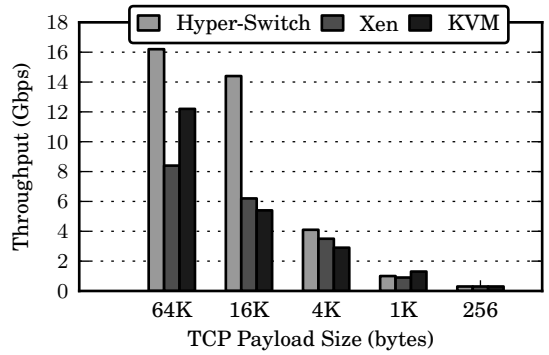

Figure 4: *Throughput results from TCP stream traffic between a single pair of VMs under different payload sizes.*

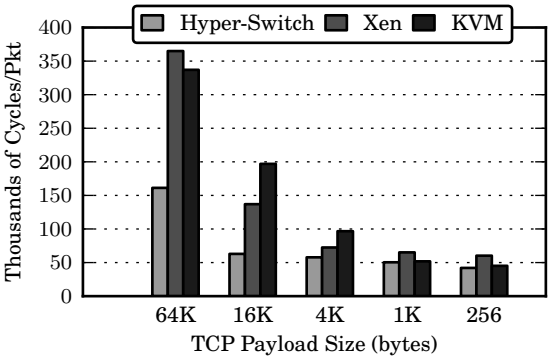

Figure 5: *CPU load results from TCP stream traffic between a single pair of VMs under different payload sizes.*

mended practice is to dedicate cores for running the driver domain's vCPUs. In this evaluation, the driver domain was configured with up to 8 vCPUs.

Open vSwitch under KVM. In KVM, Open vSwitch is implemented entirely in the hypervisor (also referred to as the *KVM host*). Under KVM's *vhost-net* architecture, all network packets are forwarded to the vhost-net driver in the host, which is similar to Xen's netback. But unlike netback, there is a separate vhost-net (kernel) thread for every vNIC in the system. The vhost-net's threads can also be run on dedicated cores.

5.2 Experimental Results

5.2.1 Inter-VM Performance and Scalability

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

Single VM Pair. In the first set of experiments, traffic was set up between just a single pair of VMs. Each guest VM's vCPU was pinned to a separate core within the same processor node to avoid any potential VM scheduling effects. 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. 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.

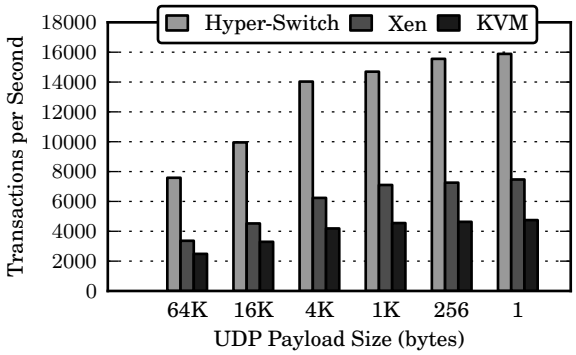

Figure 6: *Latency results from UDP request/response traffic between a single pair of VMs under different payload sizes.*

First, as shown in Figure 4, 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, 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 transmitting VM was the performance bottleneck. Figure 5 shows the average CPU load (cycles/packet) in each of these experiments. Clearly, the Hyper-Switch is more efficient in processing packets than both the existing architectures in KVM and Xen.

Second, as shown in Figure 6, higher transactions per second was achieved under Hyper-Switch, across all UDP payload sizes, with request-response 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 the Hyper-Switch among all the three architectures. On average, the transactions per second under Hyper-Switch was ∼117% higher than that under Xen and ∼222% higher than that under KVM. So the Hyper-Switch architecture is suited for both bulk as well as 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 sub-

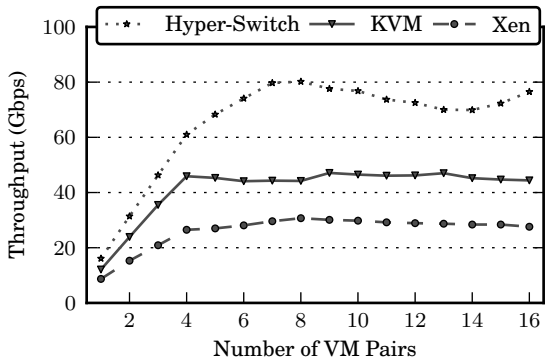

Figure 7: *Pairwise performance scalability results. Multiple concurrent TCP streams set up between pairs of VMs. Figure shows aggregate inter-VM throughput as the number of VM pairs is increased.*

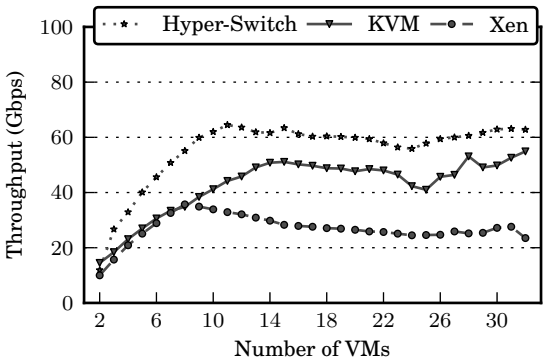

Figure 8: *All-to-all performance scalability results. Multiple concurrent TCP streams set up between all VMs. Figure shows aggregate inter-VM throughput as the number of VMs is increased.*

sequent experiments. Again, the guest VMs' vCPUs, the vhost-net kernel threads (KVM), and the driver domains' vCPUs (Xen) were pinned to specific processor cores. Also, the VMs that were communicating with each other were always pinned to the same processor node. The pinning was done such that, in each experiment, the load was uniformly distributed across all the processor modules and nodes in the system. For instance, one core from each module was used for pinning VMs across the system, before the other cores were used. Under KVM, the guest VMs' vCPUs and the vhost-net kernel threads were pinned. As a result, when the system was scaled beyond 8 pairs of VMs, each processor core had to run one of the guest VM's vCPUs *and* one of the vhost-net threads. Under Xen, the driver domain was configured with 8 vCPUs. The driver domain's vCPUs were distributed 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.

The results in Figure 7 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 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. On average, the throughput under Hyper-Switch was ∼55% higher than that under KVM and ∼146% higher than that under Xen. Figure 7 also shows three distinct regions in Hyper-Switch's performance curve: (1) The performance scaled almost linearly, from ∼16.2 Gbps to ∼62.7 Gbps, between 1 and 4 pairs of VMs. (2) The performance continued to scale linearly but at a lower rate, from ∼62.7 Gbps to ∼81 Gbps, between 5 and 7 pairs of VMs. (3) The performance did not scale beyond 8 pairs of VMs.

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. But there are limits to how many packets that can be processed by a single processor. This is determined in part by the underlying hardware architecture. The hardware 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 [25]. 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. When choosing an idle core for offloading packet processing, preference is given to idle processor cores that are on the same node as the receiving VM's vCPU, to take advantage of any CPU cache locality. 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, to avoid potential cache interference effects [3].

In the first region of the curve (Figure 7), each pair of VMs were run in a separate processor node. So the packet processing could be offloaded to other cores

within the same node. As a result, 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 CPU caches. So, effectively, all these factors offset the increase in packet processing concurrency and hence the performance stopped scaling.

All-to-all Scalability Experiments. In the second set of scalability experiments, TCP stream-based traffic was set up between every pair of VMs in the system in both 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. The configuration and setup was similar to the previous set of experiments.

Figure 8 shows the results from these 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. 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.

5.2.2 External Performance

In the external experiments, the network traffic was set up between guest VM(s) and the external client. The driver domain, under Hyper-Switch and Xen, was configured with only a single vCPU. In the TX and RX experiments, there were one or two guest VMs (concurrently) sending and receiving packets respectively. The guest VMs' vCPUs and the driver domain's vCPU were again pinned.

The results from these experiments showed that the Hyper-Switch's performance was comparable and in some cases even better than the performance under KVM and 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. 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 (7.5 Gbps) and KVM (7.8 Gbps) than Xen (4.1 Gbps). But with two guest VMs receiving packets, line rate of ∼9.4 Gbps was achieved under both 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. 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.

TCP request-response traffic was also set up between a single guest VM and the external client. In these experiments, the Hyper-Switch achieved 13,243 transactions per second of as compared to 10,721 and 11,342 under KVM and Xen respectively. 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.

5.2.3 Design Evaluation

Experiments were also run to determine the *offload criteria* under Hyper-Switch. In these experiments, the packet processing was offloaded to different CPU cores relative to where the transmitting and receiving VMs' vCPUs were running. The results from these experiments[5] indicated that, for best performance, packet processing must be offloaded to a processor module where both the cores were idle. Further, while searching for idle modules, first the processor node on which the receiving VM's vCPU was running must be searched, before searching the other nodes in the system. However, the offload criteria could vary depending on a processor's cache hierarchy. So the exact criteria must be determined based on the particular hardware platform on which Hyper-Switch is run.

6 Related Work

The current state-of-the-art network subsystem architectures for virtualized servers can be broadly classified

[5]Due to lack of space, the results from these experiments are not presented here.

into three categories. The first category of systems includes a simple network card (NIC) that is virtualized by a software intermediary, either the hypervisor (e.g. KVM [26], VMware ESX server [10]) or a driver domain (e.g. Xen [13]). 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 [1], VMware vswitch [32], Cisco Nexus 1000v [8], Open vSwitch [24], etc.—that are used in these systems. Recently, Rizzo *et al*. [30] also proposed a new virtual switching solution based on their netmap API. They use memory-mapped buffers to avoid data copies inside the host. It will be interesting to see if the netmap API can be exported all the way to the VMs. Unlike Hyper-Switch, all these existing systems implement the entire virtual switch within a single software domain—either the hypervisor or the driver domain. But we believe that the optimizations proposed in this paper are applicable to many of these solutions. Further, in this paper, the Hyper-Switch's performance was only compared to 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 [33]. Unfortunately, it is hard to compare this to Hyper-Switch's performance since the hardware platforms used in the evaluations are vastly different.

The second category of systems employ more sophisticated NICs (direct-access NICs) with multiple *contexts* that present a vNIC interface directly to each VM [20, 27, 34]. Today, there exists an industry-wide standard called SR-IOV, which has been adopted by several network interface vendors to implement this solution [6, 16, 22]. These NICs also implement a virtual switch internally within the hardware. However, today most of them only implement a rudimentary form of switch. The sNICh [28] architecture explores the idea of switch/server integration. 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*. [19] propose offloading Open vSwitch's in-kernel data path to programmable NICs. Similarly, one can also imagine offloading Hyper-Switch's data plane to the NIC hardware. These solutions can enable high-performance since the VMs directly communicate with the NIC. But, in general, they lack the flexibility that pure software solutions offer.

The third category of switches attempt to leverage the functionality that already exist in today's datacenter switches. This approach uses an external switch for switching *all* network packets [9, 23]. But, fundamentally, this approach results in a waste of network bandwidth since even packets from inter-VM traffic must travel all the way to the external switch and back again.

There have also been proposals to distribute virtual networking across all endpoints within a data center [7, 11]. 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. All these architectures are aimed at solving the network management problem, which is not the focus of this paper. But the Hyper-Switch can easily be a part of these solutions.

7 Conclusions

This paper presented the Hyper-Switch architecture that combines the best of the existing last hop virtual switching 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 the fast, efficient data plane of a flow-based software switch. The driver domain is needed only for handling external network traffic.

Further, this paper also presented several carefully designed optimizations that enable efficient packet processing, better utilization of the available CPU resources, and higher concurrency. The optimizations take advantage of the Hyper-Switch data plane's integration within the hypervisor. As a result, the Hyper-Switch enables much improved and scalable network performance, while maintaining the robustness and fault tolerance that derive from the use of driver domains. Moreover, these optimizations should be a part of any virtual switching solution that aims to deliver high performance.

This paper also presented an evaluation of the Hyper-Switch architecture using a prototype implemented in the Xen platform. The evaluation showed that, for inter-VM network communication, the Hyper-Switch achieved higher performance and exhibited better scalability than both Xen's default network I/O architecture and KVM's vhost-net architecture. Further, the external network performance under Hyper-Switch was comparable and in some cases even better than the performance under Xen and KVM.

References

[1] Linux Ethernet bridge. `http://www.linuxfoundation.org/collaborate/workgroups/networking/bridge`.

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

[3] AMD CORPORATION. Shared level-1 instruction-cache performance on AMD family 15h CPUs.

[4] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T. L., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles* (October 2003), pp. 164–177.

[5] BENSON, T., AKELLA, A., AND MALTZ, D. A. Network traffic characteristics of data centers in the wild. In *IMC* (2010).

[6] BROADCOM CORPORATION. BCM57712 product brief. `http://www.broadcom.com/collateral/pb/57712-PB00-R.pdf`, January 2010.

[7] CABUK, S., DALTON, C. I., RAMASAMY, H., AND SCHUNTER, M. Towards automated provisioning of secure virtualized networks. In *CCS '08: Proceedings of the 15th ACM Conference on Computer and Communications Security* (October 2007), pp. 235–245.

[8] 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.

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

[10] DEVINE, S., BUGNION, E., AND ROSENBLUM, M. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. *US Patent #6,397,242* (October 1998).

[11] EDWARDS, A., FISCHER, A., AND LAIN, A. Diverter: A new approach to networking within virtualized infrastructures. In *WREN '09: Proceedings of the ACM SIGCOMM Workshop: Research on Enterprise Networking* (August 2009).

[12] ESMAEILZADEH, H., BLEM, E., ST. AMANT, R., SANKARALĪNGAM, K., AND BURGER, D. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on Compute r architecture* (New York, NY, USA, 2011), ISCA '11, ACM, pp. 365–376.

[13] FRASER, K., HAND, S., NEUGEBAUER, R., PRATT, I., WARFIELD, A., AND WILLIAMS, M. 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).

[14] GREENBERG, A., HAMILTON, J., MALTZ, D. A., AND PATEL, P. The cost of a cloud: Research problems in data center networ ks. *SIGCOMM Computer Communcation Review 39*, 1 (2009), 68–73.

[15] INTEL. `http://goo.gl/5lpY8`, 2010. "Intel 1000 Core Chip".

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

[17] KUMAR, S., RAJ, H., SCHWAN, K., AND GANEV, I. 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).

[18] LANDAU, A., BEN-YEHUDA, M., AND GORDON, A. SplitX: Split guest/hypervisor execution on multi-core. In *WIOV '11: Proceedings of the 4th Workshop on I/O Virtualization* (May 2011).

[19] LUO, Y., MURRAY, E., AND FICARRA, T. 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).

[20] MANSLEY, K., LAW, G., RIDDOCH, D., BARZINI, G., TURTON, N., AND POPE, S. Getting 10 Gb/s from Xen: Safe and fast device access from unprivileged domains. In *Proceedings of the Euro-Par Workshop on Parallel Processing* (August 2007), pp. 224–233.

[21] MENON, A., SANTOS, J. R., TURNER, Y., JANAKIRAMAN, G., AND ZWAENEPOEL, W. Diagnosing performance overheads in the Xen virtual machine environment. In *VEE '05: Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments* (June 2005), pp. 13–23.

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

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

[24] PFAFF, B., PETTIT, J., KOPONEN, T., AMIDON, K., CASADO, M., AND SHENKER, S. Extending networking into the virtualization layer. In *HotNets-VIII: Proceedings of the Workshop on Hot Topics in Networks* (October 2009).

[25] PORTERFIELD, A., FOWLER, R., MANDAL, A., AND LIM, M. Y. Empirical evaluation of multi-core memory concurrency. Tech. rep., RENCI, January 2009. `www.renci.org/publications/techreports/TR-09-01.pdf`.

[26] QUMRANET. KVM: Kernel-based virtualization driver. `http://www.redhat.com/f/pdf/rhev/DOC-KVM.pdf`.

[27] RAJ, H., AND SCHWAN, K. 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).

[28] RAM, K. K., MUDIGONDA, J., COX, A. L., RIXNER, S., RANGANATHAN, P., AND SANTOS, J. R. sNICh: Efficient last hop networking in the data center. In *ANCS '10: Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (October 2010), pp. 1–12.

[29] RAM, K. K., SANTOS, J. R., TURNER, Y., COX, A. L., AND RIXNER, S. 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* (March 2009), pp. 61–70.

[30] RIZZO, L., AND LETTIERI, G. VALE, a switched ethernet for virtual machines. In *CoNEXT '12: Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies* (Decemeber 2012), pp. 61–72.

[31] SANTOS, J. R., TURNER, Y., JANAKIRAMAN, G., AND PRATT, I. Bridging the gap between software and hardware techniques for I/O virtualization. In *ATC '08: Proceedings of the USENIX Annual Technical Conference* (June 2008), pp. 29–42.

[32] VMWARE, INC. VMware virtual networking concepts. `http://www.vmware.com/files/pdf/virtual_networking_concepts.pdf`, 2007.

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

[34] WILLMANN, P., SHAFER, J., CARR, D., MENON, A., RIXNER, S., COX, A. L., AND ZWAENEPOEL, W. Concurrent direct network access for virtual machine monitors. In *HPCA '07: Proceedings of the 13th International Symposium on High-Performance Computer Architecture* (February 2007).