# Hyper-Switch: A Scalable Software Virtual Switching Architecture

KAUSHIK KUMAR RAM, ALAN L. COX, AND SCOTT RIXNER

Kaushik Kumar Ram recently graduated from Rice University with a Ph.D. in Computer Science. In his graduate research work, he explored new mechanisms and architectures for the network subsystem in virtualized systems. He likes to build systems software to solve interesting problems in the areas of operating systems and networking. He received his B.Tech in Computer Science and Engineering from Indian Institute of Technology in Guwahati, India. kaukum@gmail.com

Alan L. Cox is an Associate Professor of Computer Science at Rice University and a long-time contributor to the FreeBSD project. Over the years, his research has sought to address fundamental problems at the intersection of operating systems, computer architecture, and networking. Prior to joining Rice, he earned his B.S. at Carnegie Mellon University and his Ph.D. at the University of Rochester. alc@rice.edu

Scott Rixner is an Associate Professor of Computer Science at Rice University. His research focuses on the interaction between operating systems, runtime systems, and computer architectures; memory controller architectures; and hardware and software architectures for networking. He works with both large server-class systems and small embedded systems. Prior to joining Rice, he received his Ph.D. from MIT. rixner@rice.edu

In virtualized datacenters, the last hop switching happens inside a server. In this article we describe the Hyper-Switch, a highly efficient and scalable software-based network switch that works alongside driver domains. Hyper-Switch outperforms existing virtual switches used in Xen and KVM, especially for inter-VM network traffic, and this performance will soon be critical in datacenters.

## Machine Virtualization in Datacenters

Machine virtualization has become a cornerstone of modern datacenters; it enables server consolidation as a means to reduce costs and increase efficiencies. Many cloud-based service infrastructures 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. These benefits of machine virtualization are now widely recognized. Consequently, the number of virtual servers in production is rapidly increasing.

The use of machine virtualization has led to considerable change to the datacenter network. 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 inside the physical server. In other words, a virtual switch within the server is ultimately responsible for demultiplexing and forwarding packets to their destinations.

Communication between servers within the same datacenter already accounts for a significant fraction of a datacenter's total network traffic [3]. Moreover, a recent study of multiple datacenter networks reported that 80% of the traffic originating at servers in cloud datacenters never leaves a rack [1]. Further, 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 [2]. If this prediction comes to pass, then a rack of servers may be replaced by VMs in a single physical server, and the network traffic that today never leaves a rack may instead never leave a server. These datacenter trends necessitate the need for a high-performance virtual switch to support efficient communication—especially between VMs—in virtualized servers.

## Software Virtual Switching Solutions

There are many I/O architectures for network communication in virtualized systems. Of these, software device virtualization is 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 used to access the physical devices; they provide a safe execution environment for the device drivers.

Arguably, 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. This is becoming an important requirement, especially as servers in datacenters move toward multi-tenancy; however, this reliability comes at a price because the use of driver domains

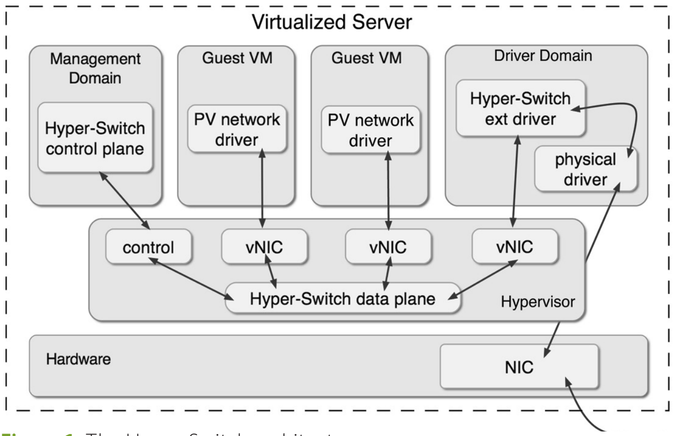

**Figure 1:** The Hyper-Switch architecture

leads to significant software overheads that not only reduce the achievable I/O performance but also severely limit I/O scalability [8]. Specifically, the sharing of I/O buffers between the driver domain and guest VMs is expensive because it requires hypervisor intervention to maintain memory isolation.

There are fundamental problems with traditional driver domain architectures. Essentially, the driver domain must be scheduled to run whenever packets are waiting to be processed. 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, which is very hard to achieve for all workloads.

In real-world virtualization deployments , dedicating processor cores to the driver domain is standard practice . 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 (e.g., SplitX [4]); however, this can lead to underutilization of these cores. This goes against one of the fundamental tenets of virtualization: to enable the most efficient utilization of the server resources.

## Hyper-Switch

We explored the virtual switching design space to see whether we could achieve both high-performance and fault tolerance at the same time. If you look at 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 these components are implemented inside a driver domain in Xen and the host OS in KVM. This colocation is purely a matter of convenience because packets must be switched when they are moved between the virtual devices and the device drivers.

We introduce the Hyper-Switch [7], 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 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.

Figure 1 illustrates the Hyper-Switch architecture. In Hyper-Switch, 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. Incoming external network traffic is initially handled by the driver domain because it hosts the device drivers, and then is forwarded to the destination VM through Hyper-Switch. For outgoing external traffic, these two steps are reversed. 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 is similar to how switching is performed using OpenFlow [5].

### *Basic Design*

Packet processing by 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. Packet processing proceeds in four stages:

1. **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.
2. **Packet switching.** In the second stage, the packet is switched to determine its destination. Switching is triggered by a hypercall from the transmitting VM and begins with reading the transmit ring to find new packets. Each packet is then pushed to Hyper-Switch's data plane where it is switched using the flow-based approach. The data plane must be able to read the packet's headers in order to switch it. Because 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.
3. **Packet copying.** In the third stage, the switched packet is copied into the receiving 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, the VM copies the packet into its memory. Note that the packet is copied directly from the transmitting VM's memory to the receiving VM's memory.

4. **Packet reception.** In the fourth and final stage, the paravirtualized network driver in the destination VM pushes the newly received packet into its network stack, where the packet is processed and eventually handed to some application. Note that the destination VM is already notified in the previous stage. So packet reception can happen as soon as the hypercall for copying the packet is complete.

### *Optimizations*

Another important contribution of this work is a set of optimizations that increase performance. They enable Hyper-Switch to support both bulk and latency sensitive network traffic efficiently. They include:

- **Preemptive packet copying.** Packet copies are performed by default in a receiving VM's context; however, delivering a notification to a VM already requires entry into the hypervisor. So 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.
- **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. To mitigate these overheads, 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 of whether the host processor is busy or not. But, unlike those devices, 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.
- **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 all the internal receive queues for packets waiting to be copied and polling all the transmit rings for packets waiting to be switched. This can be performed by processor cores that are currently idle. In this scheme, packet copying is prioritized over switching because packet copying is typically the more expensive operation, and a receiving VM is more likely to be performance bottlenecked than a transmitting VM.

  The idle cores are woken up just when there is work to be done. On the receive side, this can be ascertained precisely when switched packets are pending to be copied at a VM. Then one of the idle cores is chosen and woken up to perform the packet copies. A low-overhead mechanism is used to offload work to the idle cores. Note that this mechanism neither involves the scheduler nor requires any context-switching; instead, it uses a simple interprocessor messaging facility to directly request a specific idle core to copy packets to the VMs. Also, this mechanism attempts to spread the work across many idle cores to increase concurrency. Further, the offload mechanism is tuned to take advantage of CPU cache locality.

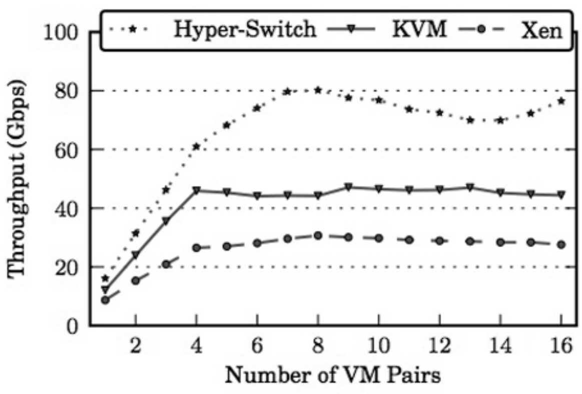

**Figure 2:** Pairwise performance scalability results

These optimizations enable efficient packet processing, better utilization of the available CPU resources, and higher concurrency. In particular, they take advantage of Hyper-Switch data plane's integration within the hypervisor and its proximity to the scheduler. As a result, Hyper-Switch enables much improved and scalable network performance, while maintaining the robustness and fault tolerance that derive from the use of driver domains.

## Evaluation

We built a prototype of the Hyper-Switch architecture in the Xen virtualization platform. Here the switch's data plane was implemented by porting parts of Open vSwitch [6] to the Xen hypervisor. Open vSwitch's control plane was used without modification. We also developed a new paravirtualized 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.

Then we evaluated Hyper-Switch using this 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. To achieve this, 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. The evaluation showed that Hyper-Switch's performance was superior in terms of absolute bandwidth as well as scalability as the number of VMs and traffic flows were varied. Figure 2 shows the results from the pairwise scalability experiments, where the number of VM pairs was scaled up. Here, on a 32-core AMD machine, Hyper-Switch achieved a peak net throughput of ~ 81 Gbps as compared to only ~ 31 Gbps and ~ 47 Gbps under Xen and KVM, respectively. Interested readers are referred to our USENIX publication that includes more results from the evaluation [7].

## Conclusion

In this work, we designed Hyper-Switch, which combines the best of the existing last hop virtual switching architectures. It hosted 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 implemented just the fast, efficient data plane of a flow-based software switch. The driver domain was needed only for handling external network traffic.

We also implemented several carefully designed optimizations that enabled efficient packet processing, better utilization of the available CPU resources, and higher concurrency. As a result, the Hyper-Switch enabled much improved and scalable network performance, while maintaining the robustness and fault tolerance that derives from the use of driver domains. We believe that these optimizations should be a part of any virtual switching solution that aims to deliver high performance. The Hyper-Switch architecture demonstrates that it is feasible to switch packets between VMs at high-speeds without sacrificing reliability.

### *References*

[1] T. Benson, S. Akella, and D. A. Maltz, "Network Traffic Characteristics of Data Centers in the Wild," IMC (2010).

[2] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11 (ACM, 2011), pp. 365-376.

[3] 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, no. 1 (2009), pp. 68-73.

[4] A. Landau, M. Ben-Yehuda, and A. Gordon, "SplitX: Split Guest/Hypervisor Execution on Multi-Core," WIOV '11: Proceedings of the 4th Workshop on I/O Virtualization (May 2011).

[5] 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, no. 2 (April 2008), pp. 69-74.

[6] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker, "Extending Networking into the Virtualization Layer," HotNets-VIII: Proceedings of the Workshop on Hot Topics in Networks (October 2009).

[7] K. K. Ram, A. Cox, M. Chadha, and S. Rixner, "Hyper-Switch: A Scalable Software Virtual Switching Architecture," ATC '13: Proceedings of the USENIX Annual Technical Conference (June 2013).

[8] 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," *VEE '09: Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (March 2009), pp. 61-70.