Network I/O Fairness in Virtual Machines

Muhammad Bilal Anwer, Ankur Nayak, Nick Feamster, Ling Liu
School of Computer Science, Georgia Tech

ABSTRACT

We present a mechanism for achieving network I/O fairness in virtual machines, by applying flexible rate limiting mechanisms directly to virtual network interfaces. Conventional approaches achieve this fairness by implementing rate limiting either in the virtual machine monitor or hypervisor, which generates considerable CPU interrupt and instruction overhead for forwarding packets. In contrast, our design pushes per-VM rate limiting as close as possible to the physical hardware themselves, effectively implementing per-virtual interface rate limiting in hardware. We show that this design reduces CPU overhead (both interrupts and instructions) by an order of magnitude. Our design can be applied either to virtual servers for cloud-based services, or to virtual routers.

Categories and Subject Descriptors: C.2.1 [Computer-Communication Networks]: Network Architecture and Design C.2.5 [Computer-Communication Networks]: Local and Wide-Area Networks C.2.6 [Computer-Communication Networks]: Internetworking

General Terms: Algorithms, Design, Experimentation, Performance

Keywords: Network Virtualization, NetFPGA, Xen

1. Introduction

An important aspect of virtual machine design and implementation is *fairness* in resource allocation across virtual machines. Although it is relatively well understood how to fairly allocate computing resources like CPU cycles, the notion of fairness becomes less clear when it is applied to I/O—and, in particular, network I/O. A common approach for controlling I/O resource utilization is to implement scheduling in the virtual machine monitor or hypervisor. Unfortunately, this approach induces significant CPU overhead due to I/O interrupt processing.

Various approaches to increase the performance of virtual-machine I/O have been proposed. Some try to provide new scheduling algorithms for virtual machines for better network I/O performance. Other techniques use existing schedulers and try to provide system optimizations, both in software and in hardware. From the operating system perspective, they fall into two categories: the driver-domain model and those that provide direct I/O access for high speed network I/O.

A key problem in virtual machines is *network I/O fairness*, which guarantees that no virtual machine can have disproportionate use of the physical network interface. This paper presents a design that achieves network I/O fairness across virtual machines by applying rate limiting in hardware on virtual interfaces. We show that applying rate limiting in hardware can reduce the CPU cycles required to implement per-virtual machine rate limiting for network I/O. Our design applies generally to network I/O fairness for network interfaces in general, making our contributions applicable to any setting where virtual machines need network I/O fairness (i.e., either for virtual servers [9] or virtual routers [20]).

Our goals for the implementation of network I/O fairness for virtual machines are four-fold. First, the rate limiter should be *fair*: it should be able to enforce network I/O fairness across different virtual machines. Second, the mechanism must be *scalable*: the mechanism must scale as the number of virtual machines increases. Third, it must be *flexible*: because virtual machines may be continually remapped to the underlying hardware, the mechanism must operate in circumstances where virtual ports may be frequently remapped to underlying physical ports; this association should also be easily *programmable*. Finally, the mechanism must be *robust*, so that neither benign users nor malicious attackers can subvert the rate-control mechanism.

We achieve these design goals with a simple principle: *push rate limiting as close as possible to the underlying physical hardware, suppressing as many software interrupts as possible*. Our implementation builds on our previous work in fast data planes for virtual routers that are built on commodity hardware [8]. The design suppresses interrupts by implementing rate limiting for each virtual queue in hardware, preventing the hypervisor or operating system from ever needing to process the packets or implement any fairness mechanisms. Our evaluation shows that implementing rate limiting directly in hardware, rather than relying on the virtual machine hypervisor, can reduce both CPU interrupts and the required number of instructions to forward packets at a certain rate by an order of magnitude.

The rest of this paper proceeds as follows. Section 2 provides background and related work on network virtualization in virtual machine's context. Section 4 presents the basic design of a virtualized network interface card to enable network resource isolation in hardware and freeing CPU from unwanted packets overhead to do better virtual machine resource scheduling; this design is agnostic to any specific programmable hardware platform. Section 5 presents an implementation of our design using the NetFPGA platform. Sec-

VISA'10, September 3, 2010, New Delhi, India.

tion 6 presents a preliminary evaluation of the virtualized network interface card; Section 7 discusses limitations and possible extensions of the current implementation. Section 8 concludes with a summary and discussion of future work.

2. Background and Related Work

Many CPU scheduling algorithms in virtual machines follow process scheduling algorithms from operating systems. For example, Xen [9] has implementations of Borrowed Virtual Time (BVT) and Simple Earliest Deadline First (sEDF). Recent work [10, 15] has shown that CPU schedulers do not perform well in the presence of a combination of I/O intensive and CPU intensive applications running together on a single server. It can result in unfair scheduling of resources for CPU intensive or network I/O intensive virtual machines. Virtual machine monitors must maintain both network I/O fairness and CPU resource fairness in a virtualized environment while providing the best performance for both CPU-intensive and network-intensive applications. To enable fair sharing of CPU and network resources, we offload the task of network virtualization and fairness from the CPU and enforce it in the virtualized network interface card.

Network I/O virtualization has been used in both OS virtualization and a non-OS virtualization context. Virtual Interface Architecture (VIA) [12] is an abstract model that is targeted towards system area networks and tries to reduce the amount of software overhead compared to traditional communication models. In traditional models, the OS kernel multiplexes the access to hardware peripherals, so all communication must trap within the kernel. VIA tries to remove this communication overhead by removing the kernel operations in each communication operation. Similarly, Remote Direct Memory Access (RDMA) [4] is a data communication model that allows the network interface card direct memory access to application memory without copying data between kernel and user space.

In OS virtualization, the *driver-domain model* and the *direct I/O model* are two approaches for achieving network I/O performance while maintaining fairness. The driver-domain model [21, 23] uses a separate virtual machine for device drivers. This domain runs a mostly unmodified operating system and simplifies the problem of providing device drivers to many virtual machines. The driver domain provides better fault tolerance by isolating driver failures to a separate domain compared to maintaining device drivers in hypervisor. On the other hand, the direct I/O model gives direct hardware access to the guest domains running on the server [6, 7, 17, 22]. This approach provides near-native performance, but it sacrifices fault isolation and device transparency because it breaks the driver-domain model.

These two approaches [6, 7] provide separate queues to each virtual machine. They offload the load from the Xen software and perform multiplexing and demultiplexing in hardware. Mansley *et al.* extend the netback and netfront architecture of Xen to allow domUs to opt for *direct I/O* or "fast path" instead of driver domain or "slow path" [17]. CDNA (Concurrent, Direct Network Access) goes further by combining hardware multiplexing and demultiplexing [22].

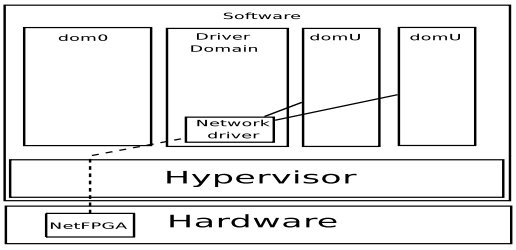

Figure 1: Xen Separate Driver Domain.

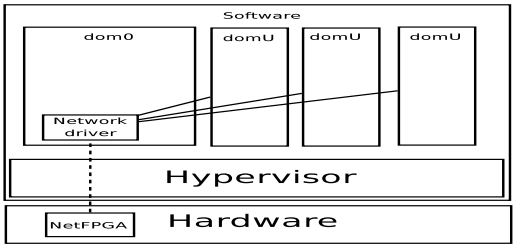

Figure 2: Dom0 as Driver Domain.

It also assigns each virtual machine to a queue and bypasses the driver domain to provide direct network traffic I/O.

Many virtual machine monitors (e.g., Xen [9], Microsoft Hyper-V [2] and L4 [16]) use the driver-domain model to provide virtualization of I/O devices. In Xen, the driver domain can be run separately or in domain 0, as shown in Figures 1 and 2, respectively. This model provides a safe execution environment for the device drivers, which enables improved fault isolation.

Both of these approaches have various problems. Although direct I/O provides better performance, it can cause reliability problems [11]. The driver-domain model is preferred because a separate domain can handle device failures without affecting the guest domains [24]. Direct I/O also requires the guest OS to have the device-specific driver, which increases the complexity of the guest OS and thus makes porting and migration more difficult. The driver-domain model, on the other hand attempts to keep the guest OS simple, but it does suffer from performance overhead, as the driver domain becomes a bottleneck, since every incoming packet has to be inside the driver domain before it can be copied to the guest domain.

In both *driver-domain model* and *direct I/O model*, interrupts are received by *hypervisor*, which dispatches them to the driver domain in case of driver domain model or to the guest domains in case of direct I/O model. In this work we preserve the driver-domain model and assign virtual queues to every virtual machine running on the server. Although this approach risks exposing the the hypervisor to unwanted interrupts, but we maintain fairness in hardware and also suppress interrupts before sending them to the hypervisor.

3. Design Goals

This section outlines the design goals and challenges for the virtualized network interface card that provides both network I/O fairness that host multiple virtual machines.

1. **Fair.** Our main goal is to provide network I/O fairness to all users who share a single physical server. The design should be scalable enough to provide both transmit and receive-side fairness to all users on a single server. Fair access to the network resources enables a cloud service provider to allocate a fixed amount of bandwidth to each user at the server level, so that as load increases, no user should have access to an unfair share of resources.
2. **Scalable.** Increasing CPU processing power [5] with increasing interconnection bandwidths (e.g, PCIe 2.0) means that data can be sent to the CPU at increasingly high rates. These two facts point towards a higher per-server network bandwidth in cloud infrastructure and data-center networks. A general rule of thumb of one virtual machine per thread per processor [19] means that a single server might host tens of virtual machines. Therefore, the design should be scalable enough to handle the bandwidth of an increasing number of virtual machines per server.
3. **Flexible.** The design should allow the physical server owner to dynamically change the physical port association of a virtual Ethernet card on the fly. A physical card might have an unequal number of physical and virtual Ethernet ports, which means that number of virtual Ethernet cards that can be maintained on a single card should not depend on physical Ethernet ports on the hardware. We assume that new virtual machines will be created regularly on a server and that they can have new MAC addresses with each new instance. Similarly, if a virtual machine migrates from one server to another, it might want to keep its MAC address as it moves; therefore, the network interface card should be able to accommodate the new MAC address if needed.
4. **Programmable.** Associating each virtual Ethernet interface should be based on MAC addresses. The virtual network interface card should be programmable enough to allow the administrator to associate the physical and virtual interfaces using a simple programmable interface. It should also enable the administrator to associate virtual Ethernet interfaces with physical ones based on user defined values (e.g, IPv4 addresses).
5. **Robust.** Design should be robust enough to not introduce any malicious behavior. e.g., Separating physical interfaces from virtual Ethernet interfaces opens the possibility of Denial of Service (DoS) attacks on the VMs on the same physical server. We aim for our design to be robust enough to withstand any malicious activity from the users residing on the same physical server.

The next sections describe the design and implementation that tries to achieve these goals.

4. Design

Our design assumes multiple users on a server sharing a physical network interface card, as shown in Figure 3.

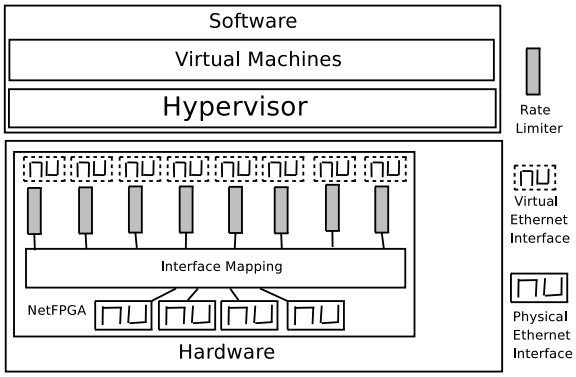

Figure 3: Virtualized network interface card with rate limiters.

The higher per-server network bandwidth requirement of cloud infrastructure and data-center networks makes scalability one of the main issues for our design. We use a CAM-based design to handle multiplexing and demultiplexing of high-speed network traffic in hardware for a large number of virtual machines.

One of our main goals is to maintain network I/O fairness in hardware and suppress unwanted interrupts on the card before sending packets to the CPU. Achieving this network I/O fairness in hardware can ultimately result in better performance and better resource scheduling for the server. Virtualized Ethernet cards available in the market [6, 7] provide a notion of virtual Ethernet interfaces, as we discuss in Section 4.1. We also explain how virtual interfaces can be mapped to the physical interfaces on a card (Section 4.2). Previous work has shown that multi-queue network interface cards provide better performance than single-queue cards [14, 21]. Because there is no open implementation of a virtualized network interface card, we first modify the NetFPGA [3] NIC implementation to provide a virtualized network interface card. We further modify this implementation to provide programmability and network I/O fairness, suppressing the interrupts inside hardware before sending them to the CPU (Section 4.3).

4.1 Virtualized Ethernet Card

Virtualized network interface cards (e.g., [6, 7]) provide packet multiplexing and demultiplexing in hardware instead of software. Without hardware support, this multiplexing and demultiplexing occurs in the driver domain; with increasing network bandwidth requirements for server, multiplexing and demultiplexing in software can easily become a performance bottleneck. Because physical Ethernet cards can have many virtual network interfaces, we implement the packet multiplexing and demultiplexing using the MAC mapping table and map each virtual network interface in VM to a physical queue in hardware.

In the NetFPGA reference design for network interface card, each Ethernet port is mapped to a corresponding Ethernet interface in software. Thus, if a packet arrives on a physical Ethernet interface, by default it is sent to the software interface in Linux kernel, at which point the user process handles the packet.

We have allocated virtual Ethernet interfaces to physical queues using a mapping table, as shown in Figure 3. Because the number of virtual Ethernet interfaces in software can be more than the physical ports available on hardware we use a table that maps physical to virtual Ethernet interface mappings so that the packet can be sent to the appropriate queue in hardware.

Each Ethernet interface must also have its own MAC address through which it can be identified to the outside world. Thus, each virtual Ethernet interface has a 48-bit register that stores the MAC address for the virtual Ethernet interface.

4.2 Mapping Ethernet Interfaces

On a physical card, the number of physical Ethernet interfaces may not be equal to virtual Ethernet interfaces. Therefore, to identify each virtual Ethernet interface uniquely, each virtual interface must have a unique MAC address. MAC addresses for the virtual Ethernet interfaces are maintained in a small table. For every incoming packet, its destination MAC address is looked up in the table to see if the packet belongs to one of the virtual Ethernet interfaces on the card. If there is a hit in this table, it means the packet is addressed to one of the virtual machines and is accepted; in case of a miss, the network interface card drops the packet.

In addition to book-keeping of MAC addresses for virtual interfaces, this table also maps each MAC address to the corresponding virtual Ethernet interface. This mapping translates the MAC address to physical queue mapping in hardware. For each incoming packet, if its destination MAC address is present in the mapping table, a table lookup returns the corresponding virtual Ethernet interface to the MAC address. Based on this value, the incoming packet is sent to the appropriate virtual interface in software.

The mapping of physical to Virtual Ethernet interfaces is dynamic and can be changed by the administrator on the fly. In addition to redirecting received packets, the mapping table has information about outgoing packets. It has a field that is looked up for every outgoing packet. This field makes sure that users can send traffic out of the physical interface through which they are allowed to send traffic out instead of any other interface.

4.3 Fairness and Interrupt Suppression

Each virtual Ethernet interface's receive queue has rate limiters placed on it. Once the packet is forwarded from the mapping tables stage, it reaches to a token bucket rate limiter as shown in figure 3. These rate limiters provide a soft limit to the traffic coming to the CPU and these limits can be changed by the administrator using a register interface through a user-space program.

Rate limiting before the receive queue of each virtual Ethernet interface can help enforce a fairness policy set by the administrator. If any of the user tries to send more traffic than what is allocated than the packets are dropped. This design ensures that different virtual machine users receive no more than their allocated share of bandwidth.

In addition to providing network traffic fairness between the virtual machines, packet dropping in hardware makes sure that no extra packets go the hypervisor. There are two reason to drop packets in hardware. First of all bandwidth

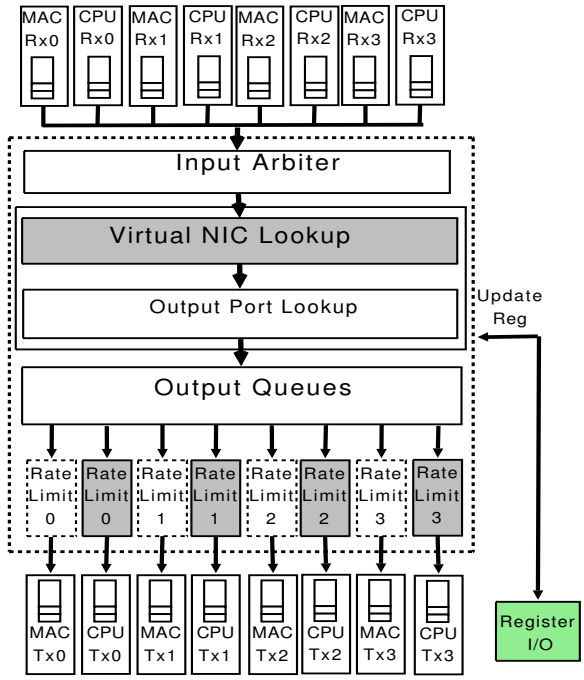

Figure 4: Pipeline for NetFPGA implementation.

of NIC interconnect interface may not be enough to send all traffic from virtualized NIC to the software e.g. PCI bandwidth is much less than 4Gbps bandwidth of NetFPGA card. Therefore NIC should not be receiving more traffic than it can push through the interconnect interface. Rate limiter imposes this limit and drops any extra packets that can not be pushed through the interconnect. Secondly, it puts a limit on bandwidth usage by each VM thus there are no extra interrupts generated for the hypervisor to handle extra traffic for a specific user.

Inter-virtual machine traffic on the same server can both be legitimate and illegitimate. Any legitimate virtual machine user on a server can mount malicious attacks on the neighbors residing on the same physical server. Its possible that a user can send a packet with his source MAC address, destined to another machine on the same server. This kind of attack will result in wasted CPU cycles for the user that is under attack. One solution can be to simply block inter-virtual machine traffic, but another solution can be to block inter virtual machine traffic based on legitimacy of inter-virtual machine traffic. A simple blocking solution mimics the behavior in servers with different NICs and can implemented by looking up the source and destination MAC addresses for each outgoing packet. If the destination MAC address and the source MAC address of packet are matched then the packet can be dropped; if there is a match for only one of the addresses, then the packet can be transmitted.

5. Implementation

We implemented this virtual NIC design with a mapping table and rate limiters for receive-side fairness for virtual machines. Through this implementation, we wanted to show

the feasibility of the design and its ability to maintain network I/O fairness and interrupt suppression.

Figure 4 shows our implementation on a NetFPGA [3] card, with the possibility of adding rate limiters on the transmit side. Figure 4 also shows the pipeline of the implementation with new modules for virtualized Ethernet card highlighted. We have used the NetFPGA reference implementation as the base implementation. We have combined "Output Port Lookup" and "Virtual NIC Lookup" stages into a single stage and have not implemented transmit-side rate limiters for the CPU queues which are shown with dotted lines in figure 4.

There are four rate limiters in hardware to provide receive-side fairness for the incoming packets to all the virtual machines running in software. Similarly, transmit-side rate limiters can be used to stop any virtual machine from sending more than its fair share (Figure 4). Because the packet generation process is handled by the CPU and each virtual machine's CPU cycle scheduling is already being done by Xen VM scheduler, it should not be possible for a VM process to get more than its share of CPU cycles allotted by scheduler and then generate high volumes of traffic. However, a user might still send traffic from unauthorized physical Ethernet interface; to counter this, we have an entry in mapping table that keeps track of outgoing traffic and prohibits any user from sending at higher than the allotted rate.

We have implemented queue mapping using a single BlockRAM-based CAM with exact matching. There are 32 entries in CAM; each entry has 3 fields. For each entry, there is a single MAC address with administrator allocated incoming and outgoing ports for it to access.

The NIC matches the destination MAC address of each packet against the CAM; if there is a hit in the table, then the card sends the packet to the corresponding CPU queue, as determined by the values that are stored in the CAM. Figure 4 shows the CPU queues after the MAC lookup stages as "CPU TxN". If there is a miss, then the packet is dropped immediately in the "Virtual NIC Lookup" stage.

For packets coming from the CPU, the interface card looks up the packet's source MAC address; if there is an entry for the source MAC address, then the interface card knows that a legitimate user sent the packet. The interface card then identifies this user's outgoing port and the packet is sent out of the allocated physical port to the particular user.

6. Results

In this section, we present the resource usage of an initial NetFPGA-based hardware implementation and the performance results.

6.1 Resource Usage

For the resource usage of our implementation we used Xilinx ISE 9.2 for synthesis. The resource usage reported here is for a design with 32-entry CAM and with four virtual interfaces on the card. Adding more virtual Ethernet interfaces will mean adding more queues on the NetFPGA card which means more resource usage. Here we are using four rate limiters on each Ethernet queue for receive side fairness therefore we only need a four entry table.

Resource	V2Pro 50 Utilization	% Utilization
Slices	15K out of 23,616	63%
Total 4-input LUTs	21.5K out of 47,232	45%
Route Thru	2K out of 47,232	4.3%
Flip Flops	15K out of 47,232	31%
Block RAMs	116 out of 232	50%
Eq. Gate Count	8,176 K	N/A

Table 1: Resource utilization for the Virtualized NIC with four rate limiters

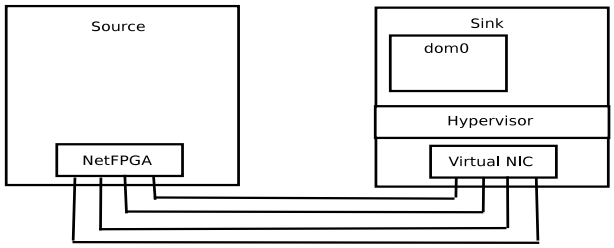

Figure 5: Experimental Setup

Our existing implementation, uses 21,500 four-input Look-Up-Tables (LUTs) of the 47,232 available. This makes 4-input LUT utilization to be 45% of available resources on the Virtex-II Pro NetFPGA. Table 1 provides various other statistics about resource usage. This resource usage will increase once we increase the number of queues in hardware and add rate limiters to those queues; it will decrease if we have less than 32 entries in the CAM.

6.2 Performance

We first show that our simple virtualized network card implementation works by assigning each virtual network interface a specific MAC address that can be used by the virtual machine. Current implementation has four queues in hardware and four interfaces in the software. In this experiment, we simply show that what is the maximum amount of traffic that can be sent from the outside servers to the virtual machines on server. Although the PCIe interface has much higher bandwidth than PCI interface, the 1x4 Gbps NetFPGA [3] card only has PCI interface; therefore, we have done measurements on NetFPGA card connected to the server through PCI interface.

Figure 5 shows our experimental setup, which consists of 2 1U servers with Intel Quad Core processors and 4GB of RAM on each machine. Each machine has one NetFPGA card. It is a single source-sink topology where source is directly connected with the sink. All four ports of the source node are connected directly to the four ports of the sink node. We used the NetFPGA-based packet generator [13] to generate high speed network traffic. On the receiver side, we had NetFPGA card with virtualized network interface card, as discussed in Section 5.

6.2.1 *Packet Demultiplexing*

We measured packet-receive rates in the Xen hypervisor to obtain a baseline measurement for forwarding speeds of the PCI-based NetFPGA NIC card. Figure 6 shows total receive rate when the packet generator floods one, two, and four

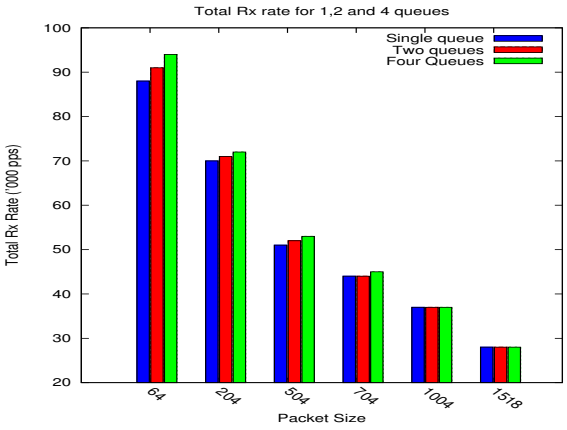

Figure 6: Total number of received packets per second at different packet sizes, for one, two and four queues

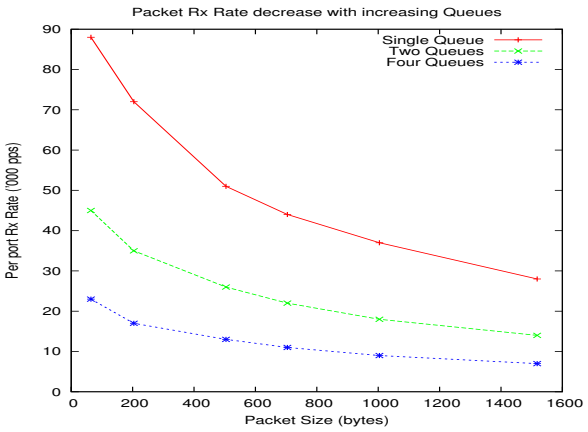

Figure 7: Number of received packets per port at different packet sizes, for one, two, and four queues.

queues of the network interface card. As we increased the number of queues from one to four, the total packet receive rate remained the same, and the bandwidth was distributed evenly across different virtual machines. Interestingly, this flooding showed fairness in the number of packets received by more than one queues. When we flooded two and four queues, the cumulative rate remained the almost same, and all queues received packets at equal rate. This shows that the virtualized NICs achieve fairness even when all hardware-based rate limiters are disabled.

Figure 7 shows the per-queue receive rate in packets per second that can be achieved when the NetFPGA card is in the PCI slot. When we increase the number of receive queues that are being flooded, the forwarding rate per port is decreased: the forwarding rate is inversely proportional to the number of queues.

These two figures show the inherent fairness in the virtualized network interface card. When all users are using the network at full capacity, traffic is equally shared among them. We are able to get equal shares for all the users mainly because of virtualized queues, and by representing each queue in software as a separate network interface. It also shows that with the PCI 32-bit version working, we can achieve a

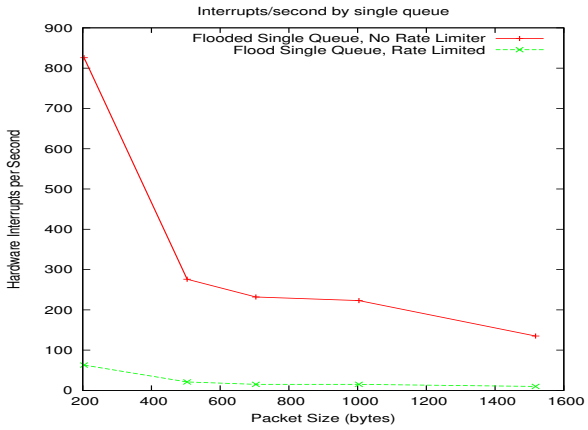

Figure 8: Reduction in Hardware Interrupts to CPU

maximum speed of about 90,000 packets per second with 64-byte sized packets. While each queue is receiving packets at approximately 23,500 packets per second.

6.2.2 *Hardware Interrupt Suppression*

As shown previously that in virtualized NIC cards, assigning separate queues to each VM, provides equal share of network traffic to each VM. But the problem comes in when we consider the number of interrupts sent to hypervisor because of traffic of each VM. If one of the four user is flooding the network and three users are using bandwidth within their limits the number of interrupts sent to hypervisor from user flooding the card will be much more higher then users staying within their limits. Here we show that by rate limiting in hardware we are effectively suppressing the user interrupts as well thus providing the hypervisor and dom0 ability to serve equal amount of CPU resources to each VM.

To measure the effectiveness of this implementation we measured the decrease in interrupts per second by decrease in packets received by the Xen Hypervisor. We measured the interrupts in hardware using open source tool Oprofile [18].

Figure 8 shows the results for this particular experiment. Here we flooded a single virtual Ethernet card with maximum possible traffic to the card. For the smallest size packets the number of interrupts per second was almost equal to the number of packets per second received by the server.

As the packet size is increases from 64 bytes to 1518 bytes, the number of interrupts per second drops. This drop in interrupts per second does not guarantee fair resource sharing on the CPU secondly for smaller packet sizes number of interrupts per second is too large.

To measure the effectiveness of the rate limiter in decreasing the number of interrupts per second, we decreased the amount of traffic received by each user to approximately 24,000 packets per second for 64B sized packets. Then we flooded the single queue with NetFPGA packet generator at approximately 1Gbps with 64B sized packets. Despite this higher rate from the packet generator, the virtualized network interface card only forwarded packets that were allowed for the particular single queue. As shown in Figure 8, this directly resulted in a decrease in the number of interrupts per second to the hypervisor.

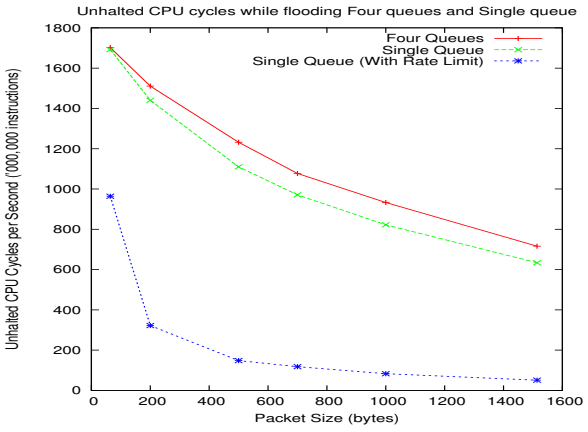

Figure 9: Unhalted CPU Cycles per second while flooding.

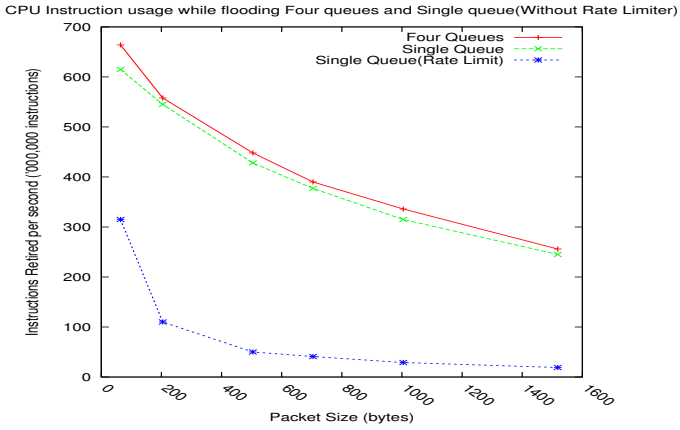

Figure 10: Instructions per second while flooding

This essentially shows that a simple rate limiter can reduce the number of interrupts to the hypervisor and dom0 and can stop a user from taking unfair advantage of CPU resource in the hypervisor and domain0 for that user.

6.2.3 *CPU Resources*

To measure the effect of excessive packets on the CPU we measured number of CPU cycles required to process each packet and instructions taken by the CPU to process those packets. For this experiment we used Xen and dom0 kernel with debug information, which resulted in overall lower packet capturing rate for the kernel. But the packet per second behavior was similar as shown in previous experiments.

We measured the number of unhalted CPU cycles on a 2.66 GHz Intel Xeon processor and number of instructions executed to capture the packets. First, we flooded all four ports of the network interface card with the network traffic and measured the number of unhalted CPU cycles per second and number of instructions executed per second, while all four queues were flooded. We repeated this process for different packet sizes.

As shown in Figure 7, the per-port receive rate decreases when we increase the number of queues being flooded. The same thing is happening here, number of CPU cycles to handle each machine's packets are distributed for different virtual machines. Figure 9 shows the number of CPU cycles being spent to serve the packets for a single queue without rate limiting, is almost equal to the number of cycles being spent when all four ports are getting equal network traffic share.

Then we limited the amount of traffic that can be sent using single virtual queue equal to 24,000 packets per second for 64-byte packets using rate limiters. After putting a limit on what traffic can be sent to a single virtual machine using the rate limiters, the number of cycles spent to serve a single queue's traffic decreased. Apart from the smallest packets, we see more than four times decrease in number of unhalted CPU cycles for larger packet sizes.

We repeated this same experiment and measured the number of instructions per second. The number of instructions spent per second with single queue being flooded was almost equal to when all four queues were bombarded using the packet generator, as shown in figure 10.

After enabling the rate limiter, we reduced the single queue's forwarding rate. Figure 10 shows the results of enabling the rate limiter and setting a lower rate limit on a single queue. We observe a decrease in the instructions per second spent by the Xen Hypervisor and dom0 for a single virtual machine user that is is more than a factor of four, for all packet sizes except 64-byte packets.

These experiments show two benefits of locating rate limiters in the virtual network interface card itself. First, using rate limiters, we can send through only that traffic to the server and to the virtual machines that they can handle in software, without excessively increasing the number of unhalted CPU cycles and instructions spent on the packets.

Second, rate limiters in the virtual network interface cards do a nice job of stopping the hypervisor and the driver domain in spending extra CPU cycles for unwanted traffic. Although the Linux traffic shaper can be used in dom0 to limit traffic to domU, using this technique will still mean that hypervisor and dom0 must receive and process the packets before discarding them. Using rate limiters can handle such unpredictability in hardware, thus allowing the Xen scheduler to more efficiently schedule the VMs themselves.

7. Future Work

Our proof-of-concept uses four queues, which is equal to the number of physical input ports available on the card. In our future work, we plan to increase the number of virtual Ethernet cards on the physical card, which can be done by increasing the number of CPU queues on the physical card, as well as making changes to the software.

All virtual network interface cards belong to a single broadcast domain. We intend to modify this constraint so that an administrator can dynamically change the broadcast domain of a network interface card. Our current implementation also has limited capabilities for collecting statistics about incoming and outgoing traffic for each virtual network interface card. network interface card. We plan to enhance our design to enable an administrator collect such statistics. We have used source and destination MAC addresses to associated queues to virtual machines. In a future design, we

intend to make this association possible by hashing on different packet fields.

In future work, we plan to both virtualize the rate limiter and make output queueing more flexible. Our current design applies separate token bucket rate limiters to each receive queue; thus, adding more virtual network interfaces also requires a larger number of rate limiters. This scaling situation becomes even worse if we want to implement transmit-side fairness using rate limiters. We are working on a solution to virtualize the rate limiter so that we can use only a single rate limiter for all queues. Our current design also does not fully exploit the resources available on the network interface card for output queues. The current design has a fixed number of output queues; if, for example, there only two users using the network interface card the, we should be able to provide bigger queues to the 2 users instead of wasting that space.

8. Conclusion

Operating system virtualization is appears in both cloud services [1] and router virtualization [20]. Until recently, however, network virtualization for virtual machines occurred entirely in software. Recent solutions achieve better performance by using hardware to multiplex and demultiplex the packets, before sending them to operating system. Moreover, network virtualization for operating systems means following existing models that have proven to be useful, such as providing fault isolation with a separate driver domain.

This paper takes a step towards providing a fast, flexible, and programmable virtualized network interface card that can provide high-speed network I/O to virtual machines while maintaining the driver domain model. In addition to providing packet multiplexing and demultiplexing in hardware for virtual machines, assigning each virtual machine to a single queue in hardware and using the network traffic rate limiters in hardware achieves network I/O fairness and maintains good performance by suppressing interrupts in hardware. This hardware-based approach reduces the interrupts that are sent to the hypervisor, as well as the number of instructions and number of CPU clock cycles spent to process each packet. Although many more functions can be added to the proposed virtual network interface card, this paper represents a first step towards providing hardware-assisted network I/O fairness for virtual machine environments.

Acknowledgments

This work was funded by NSF CAREER Award CNS-0643974 and NSF Award CNS-0626950. Ling Liu acknowledges the partial support from a NSF NetSE grant, a NSF CyberTrust grant, an IBM faculty award, and a grant from Intel research council.

REFERENCES

[1] Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2/`.

[2] Microsoft Hyper-V-Server. `http://www.microsoft.com/hyper-v-server/en/us/default.aspx`.

[3] NetFPGA. `http://www.netfpga.org`.

[4] Remote Direct Memory Access. `http://www.rdmaconsortium.org/`.

[5] Intels teraflops research chip. `http://download.intel.com/pressroom/kits/Teraflops/Teraflops_Research_Chip_Overview.pdf`, 2010.

[6] Product brief: Intel 82598 10gb ethernet controller. `http://www3.intel.com/assets/pdf/prodbrief/317796.pdf`, 2010.

[7] Product brief: Neterion x3100 series. `http://www.neterion.com/products/pdfs/X3100ProductBrief.pdf`, 2010.

[8] M. B. Anwer and N. Feamster. Building a Fast, Virtualized Data Plane with Programmable Hardware. In *Proc. ACM SIGCOMM Workshop on Virtualized Infrastructure Systems and Architectures*, Barcelona, Spain, Aug. 2009.

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*, Lake George, NY, Oct. 2003.

[10] L. Cherkasova, D. Gupta, and A. Vahdat. Comparison of the three cpu schedulers in xen. *ACM SIGMETRICS Performance Evaluation Review*, (2):42–51, Sept. 2007.

[11] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An empirical study of operating system errors. In *Symposium on Operating Systems Principles (SOSP)*, Dec. 2001.

[12] Compaq, Intel and Microsoft. *Virtual Interface Architecture Specification, Version 1.0*, 1997.

[13] G. A. Covington, G. Gibb, J. Lockwood, and N. McKeown. A Packet Generator on the NetFPGA platform. In *FCCM '09: IEEE Symposium on Field-Programmable Custom Computing Machines*, 2009.

[14] M. Dobrescu, , N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, Oct. 2009.

[15] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing performance isolation across virtual machines in xen. In *Proc. of the USENIX 7th Intl. Middleware Conference*, Feb. 2007.

[16] J. Liedtke. On -kernel construction. In *Proc. of the 15th ACM Symposium on Operating System Principles*, Dec. 1995.

[17] 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 *Euro-Par 2007 Workshops: Parallel Processing*, 2007.

[18] A. Menon, J. R. Santos, Y. Turner, G. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. In *Conference on Virtual Execution Environments (VEE)*, June 2005.

[19] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. Portland: A scalable fault-tolerant layer2 data center network fabric. In *Proc. ACM SIGCOMM*, Barcelona, Spain, Aug. 2009.

[20] Norbert Egi and Adam Greenhalgh and Mark Handley and Mickael Hoerdt and Felipe Huici and Laurent Mathy. Towards high performance virtual routers on commodity hardware. In *Proc. CoNEXT*, Dec. 2008.

[21] 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 *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual Execution Environments*, Mar. 2009.

[22] S. Rixner. Network virtualization: Breaking the performance barrier. *ACM Queue*, Jan. 2008.

[23] J. R. Santos, Y. Turner, G. J. Janakiraman, and I. Pratt. Bridging the gap between software and hardware techniques for i/o virtualization. In *USENIX Annual Technical Conference*, June 2008.

[24] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, pages 77–100, 2005.