Optimizing Network I/O Virtualization with Efficient Interrupt Coalescing and Virtual Receive Side Scaling

Yaozu Dong[1], Dongxiao Xu[1], Yang Zhang[1], Guangdeng Liao[2]
[1]Intel China Software Center [2]UC Riverside
{eddie.dong, Dongxiao.xu, yang.z.zhang}@intel.com, gliao@cs.ucr.edu

***Abstract*—Virtualization is a fundamental component in cloud computing because it provides numerous guest VM transparent services, such as live migration, high availability, rapid checkpoint, etc. However, I/O virtualization, particularly for network, is still suffering from significant performance degradation.**

In this paper, we analyze performance challenges in network I/O virtualization and observe that the conventional network I/O virtualization incurs excessive virtual interrupts to guest VMs, and the backend driver in the driver domain is not parallelized and cannot leverage underlying multi-core processors. Motivated by the above observations, we propose optimizations: efficient interrupt coalescing for network I/O virtualization and virtual receive side scaling to effectively leverage multi-core processors. We implemented those optimizations in Xen and did extensive performance evaluation. Our experimental results reveal that the proposed optimizations significantly improve network I/O virtualization performance and effectively tackle the performance challenges.

Keywords: Network I/O virtualization, Xen, Interrupt coalescing, Receive side scaling, Multi-core.

1. INTRODUCTION

Virtualization has become an integral component of the modern data center. Hypervisor, a new thin layer between the operating system (OS) and hardware platforms, provides numerous virtual machine (VM) transparent services, heavily used in large-scaled data centers [2], such as VM replication [11], rapid checkpoint [5], live migration [4], and quality of service, to guarantee service level agreement in the cloud computing environment.

In the virtualized environment, a major source of performance degradation is the cost of virtualizing network I/O devices in software to allow multiple guest VMs to share a single device in a secure manner [27]. Although hardware assisted I/O virtualization techniques, such as Single Root I/O Virtualization (SR-IOV) [8], self-assisted device [23], can reduce I/O virtualization overheads and achieve good performance, they rely heavily on extensive hardware support (both NIC devices and platform hardware) and thus significantly sacrifice flexibilities fundamental to cloud computing, such as live migration, VM replication based high availability, etc. Therefore, the software network I/O virtualization preserving high flexibilities is still a de-factor in the cloud computing environment.

Although numerous studies have been conducted to understand and improve the performance of network I/O virtualization [22][26], they are still insufficient to address the performance challenge of network I/O virtualization. In this paper, we target analyzing the remaining performance challenges of network I/O virtualization by examining Xen [2][4], the most widely-used open-source paravirtualized system. We observe that the conventional network I/O virtualization has incurred excessive virtual interrupts to guest VMs because virtual interrupts are not coalesced efficiently. Due to several rounds of trap-and-emulation, we found, in our experiments, that handling a virtual interrupt in the virtualized environment is more expensive than a physical interrupt. In addition to the excessive interrupts, we also noticed that the backend driver in the driver domain, or domain 0, is implemented as a single thread and is unable to leverage underlying multi-core processors, thus becoming a bottleneck for high speed networks.

To reduce the number of virtual interrupts, we first propose coalescing virtual interrupts. We compare frontend and backend driver coalescing, and pick up the frontend driver coalescing because it has less context switches between domain 0 and guest domains, and also does not need a high resolution timer in the software. Because network I/O virtualization introduces two layers of interrupts: physical and virtual, efficiently coordinating two interrupt coalescing policies is critical to achieve the best performance, while guaranteeing the worse case latency. Therefore, in this paper we propose an adaptive, multi-layer interrupt coalescing scheme for network I/O virtualization to dynamically throttle interrupt frequencies. In addition, we also design virtual receive side scaling. Unlike the single thread backend driver, it parallelizes the backend driver and adopts receive side scaling (RSS) to

978-0-7695-4516-5/11 $26.00 © 2011 IEEE
DOI 10.1109/CLUSTER.2011.12

bind each thread of the backend driver to a corresponding virtual CPU, to use multi-core processors effectively.

We implemented the proposed interrupt coalescing and virtual receive side scaling in Xen, and did extensive performance evaluation. The results demonstrate that coalescing virtual interrupts can reduce network I/O virtualization overhead by up to 71%, and coalescing in the frontend driver delivers better performance than coalescing in the backend driver. With the support of the adaptive multi-layer interrupt coalescing, an additional 10% CPU utilization is saved. In addition to the efficient interrupt coalescing, the virtual receive side scaling can achieve a 2.61x greater throughput than the baseline (a single thread backend driver), with 10 Gbps networking. We implemented and compared three versions of the parallelized backend driver through multiple tasklets, multiple kernel threads, and multiple high priority kernel threads. We realized that the multiple tasklet solution leads to the best performance. Although our implementation is based on Xen, the optimizations in this paper are generic enough to be applied to other virtualized systems, like KVM [16], as well.

The major contributions of this paper can be summarized as follows:

- We analyzed two challenges of the network I/O virtualization.
- We proposed efficient interrupt coalescing for network I/O virtualization and virtual receive side scaling to tackle two performance challenges.
- We implemented proposed optimizations in the real virtualized system and did extensive performance evaluation.

The rest of paper is organized as follows: In Section 2, we give a brief introduction of the Xen network I/O virtualization architecture, followed our study of the challenges in Section 3. Section 4 elaborates the proposed optimizations. We present experimental results in Section 5 and cover related literature in Section 6. Finally, we conclude the paper in Section 7.

2. XEN NETWORK VIRTUALIZATION

Paravirtualization (PV) is the paradigm of I/O virtualization in commercial hypervisors [2][16]. It shows the advantage of performance over full emulation with an efficient communication channel, such as shared memory and hypercalls to reduce hypervisor intervention to emulate port I/O and memory mapped I/O, used widely by legacy device drivers. Network I/O paravirtualization also shows its great flexibility in supervision and migration, and support in modern hypervisors and guest OSes or domains. In this paper, we focus on paravirtualized network I/O virtualization.

Figure 1 shows the architecture of paravirtualized network I/O virtualization, with Xen as the example. It runs a brand new device driver in guest OS, as a frontend driver, to work cooperatively with the service provider, named as the backend driver in domain 0 (driver domain). The backend driver runs as a delayed interrupt service, that is a tasklet in Linux which is activated when the processor returns to user space. Xen PV network I/O virtualization enables NAPI [25] for efficient usage of CPU cycles, and supports requests from multiple guests.

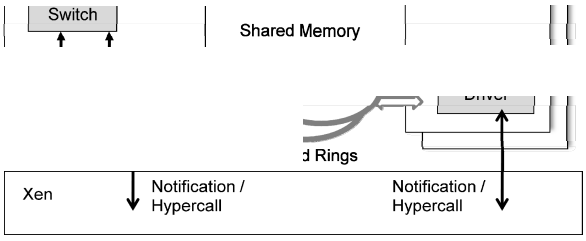


Figure 1. Xen Network I/O Virtualization

Frontend driver and backend driver communicate with virtual descriptor queues, which use shared rings and memory pages to exchange requests and responses. The frontend driver, after being pre-allocated with a receive buffer, produces requests in the shared ring, pointing to the shared memory. When a packet arrives, the physical NIC interrupts the processor and is forwarded to virtual processor of domain 0. The NIC driver (native driver in domain 0) is activated to receive the packet and send it to the virtual switch, which will dispatch the packet to the backend driver. Once the backend driver receives the packet from the guest, it consumes the request in the shared ring to get the available guest buffer, and then copies the packet to the guest buffer. It then produces a new request in the shared ring for the inbound packet, and notifies the guest, using virtual interrupts, to process the request. The frontend driver responds to the notification as an interrupt (hereafter, we use the term interrupt), and receives the packets by consuming the requests for the inbound packets in the shared ring.

3. NETWORK I/O VIRTUALIZATION CHALLENGES

Although lots of studies have been done in network I/O virtualization, to understand and improve its performance, they focused on inter-domain packet movement improvement and are insufficient to address network I/O virtualization. In this section, we discuss two major remaining challenges in network I/O virtualization: excessive virtual interrupts and single thread backend driver.

3.1. Challenge 1: Excessive Virtual Interrupts

Upon processing an inbound packet, the backend driver interrupts a guest OS immediately, indicating the packet's readiness to the guest OS. Although this scheme achieves the best response time, which is good for latency intensive workloads, it causes excessive virtual interrupts to the guest OS, thus significantly degrading I/O virtualization performance.

In the virtualized environment, the virtual interrupt processing is much more expansive than a physical interrupt. Unlike the native environment, handling a virtual interrupt in a guest OS could incur multiple rounds of trap-and-emulation, depending on which virtual

interrupt controller the guest OS implements. For instance, an I/O advanced programmable interrupt controller (APIC) masks and unmasks the servicing interrupt, by programming the I/O redirection table register. APIC programs end of interrupt (EOI) and task priority register (TPR) per interrupt. All of the above register writes may trigger trap-and-emulation in the virtualized environment. The cost of a typical trap-and-emulation of interrupt controller register operation is around 3000 to 5000 cycles, based on our experiments on Intel servers. As a result, each virtual interrupt can introduce an additional 10K cycles.

3.2. Challenge 2: Single Thread Backend Driver

Hypervisor shares the physical NIC among multiple guests through a backend driver. However, the centralized backend driver, running in a single thread, may become a bottleneck, as the number of guests goes up, or as the number of virtual receive queues in a frontend driver increases.

Figure 2 illustrates the network I/O virtualization scalability analysis in our 10 Gbps network environment (see the configuration in Section 5). It points out that the single thread backend driver services multiple guests and saturates the virtual CPU, no matter how many CPUs the domain 0 has, thus becoming a bottleneck. That is because the virtual interrupts delivered to domain 0 target virtual CPU0, which saturates the virtual CPU and threads running on top. In addition, as the number of VMs keeps increasing, the throughput drops further due to the increased overheads on context switches among VMs. Therefore, existing network I/O virtualization suffers from performance and scalability issues.

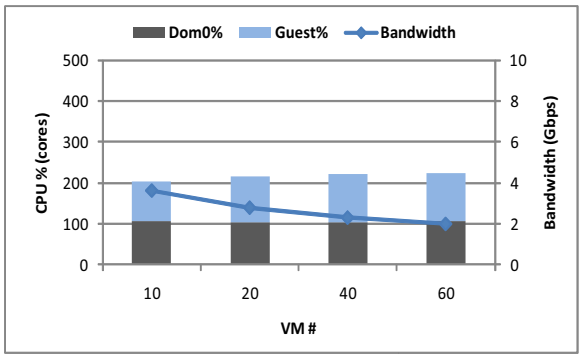


Figure 2. Network I/O Virtualization Scalability

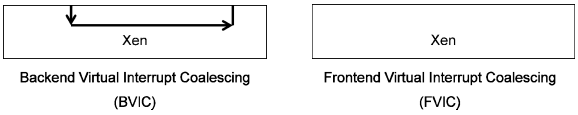
4. NETWORK I/O VIRTUALIZATION OPTIMIZATIONS

In this section, we present our optimizations for improving network I/O virtualization performance. Subsection 4.1 elaborates virtual interrupt coalescing, followed by adaptive multi-layer interrupt coalescing in Subsection 4.2. In Subsection 4.3, we present virtual receive side scaling.

4.1 Virtual Interrupt Coalescing

Interrupt coalescing is a technology to throttle interrupt frequency, to achieve the best trade off between performance and latency. As network bandwidth goes up, interrupts based on packet readiness may be too frequent. For instance, a 10 Gbps network can receive up to 0.8 million 1.5 KB packets per second, and it may have a 10X higher frequency of small packets. To reduce the number of interrupts, modern NICs leverage interrupt coalescing, achieving the best performance while maintaining the worst case latency [13].

We enhance Xen network I/O virtualization by coalescing virtual interrupts to reduce CPU cycles. There are two implementations: backend virtual interrupt coalescing (BVIC) and frontend virtual interrupt coalescing (FVIC). BVIC moderates the virtual interrupt frequency by controlling the virtual interrupt delivery in domain 0, while FVIC controls the virtual interrupt frequency fully in the guest OS, by generating a periodic timer to poll the arriving packet in the shared ring. Figure 3 shows their architectures.



Backend Virtual Interrupt Coalescing (BVIC)

Frontend Virtual Interrupt Coalescing (FVIC)

Figure 3. Virtual Interrupt Coalescing Solutions

BVIC delays interrupts from the time the earliest packet arrives. FVIC delivers interrupts based on a periodic timer, and polls the shared ring to see if there are arrival packets or not. BVIC saves additional virtual interrupts when there is no or lightweight network traffic. However, BVIC incurs more context switches between domains and also requires an additional timer in domain 0 to trigger the backend driver to deliver the coalesced interrupt. We did performance comparison of BVIC and FVIC (see Section 5) and found that FVIC achieves better performance. Therefore, in our paper, we used FVIC to coalesce virtual interrupts.

4.2 Adaptive Multi-Layer Interrupt Coalescing

Unlike the native environment, a packet in a virtualized environment traverses multi-layer network interfaces, both physical and virtual. As shown in Figure 4, the incoming packet first arrives at the physical line and driver in domain 0. Then, virtual line delivery (such as a software switch and backend driver) receives the packet and delivers it to the virtual driver in the guest OS. Because both physical and virtual device drivers implement interrupt coalescing, in the virtualized environment, packet delivery becomes multi-layer interrupt coalescing.

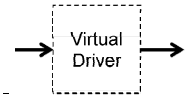


Figure 4. Network Packet Traversal in Virtualization

Multi-layer interrupt coalescing poses new challenges to virtualized systems, where latency is sensitive, such as real time systems based on CPU partitioning or pre-emptive scheduling [21]. Simply combining two independent interrupt coalescing schemes introduces additional latency to delivering a packet. This becomes worse in nested virtualization [28], where one layer of hypervisor is running on top of another layer of hypervisor. As a result, optimizing the performance of multi-layer interrupt coalescing becomes critical for network I/O virtualization.

The worst case latency of a network packet, T_{worst}, is a sum of latency introduced by the physical device driver, T_p, and the latency due to virtual device driver, T_v, as shown in Equation (1). Here, we ignore the latency due to physical line delay and software switching delay etc., because it is constant, regardless of interrupt coalescing. The CPU cycles used to process a two-layer interrupt in the worst case, C_{worst}, is a sum of cycles spent in the physical layer and the virtual layer, as shown in Equation (2). C_p, C_v stand for the CPU cycles spent per physical and virtual interrupt, and f_p, f_v represent frequency of physical and virtual interrupts. The overhead spent in the virtual interrupt side is multiplied by n, the number of guests.

$$T_{worst} = T_p + T_v \quad (1)$$

$$C_{worst} = C_p * f_p + n * C_v * f_v \quad (2)$$

Coordinating interrupt coalescing policies, between the physical device driver and the virtual device driver, is important to achieve the best performance, while guaranteeing worst case latency. The cost per interrupt handling in the physical device driver and the virtual device driver may be different and vary case by case depending on guest, driver domain (or domain 0), and their interrupt controller implementation. The ideal configuration can be calculated by Equation (3), when considering T_{worst} as a minimum value. Then, the ideal configuration can be concluded as Equation (4).

$$\frac{dC_{worst}}{dT_p} = 0 \quad (3)$$

$$T_p = T_{worst}/(1 + \sqrt{\frac{C_v}{C_p}} * \sqrt{n}) \quad (4)$$

In this paper, we propose an adaptive two-layer interrupt coalescing scheme, based on Equation (4). The cost per network interrupt, shown as C_p *and* C_v, might be hard to know, due to the vastly unpredictable facts, such as spontaneous polling from NAPI, cache hit rates, etc. We obtain those parameters from real experiments and adaptively adjust the parameters through a coalescing manager running in the driver domain (domain 0), and a coalescing client running in the guest. Figure 5 shows the architecture of our adaptive two-layer interrupt coalescing.

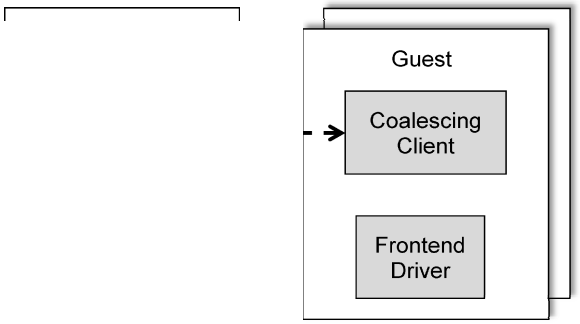


Figure 5. Adaptive Two-layer Interrupt Coalescing

4.3 Virtual Receive Side Scaling

In conventional network I/O virtualization, a single thread backend driver saturates the running CPU at ~4 Gbps, while leaving idle the rest of the CPUs in domain 0. Therefore, we multithread the backend driver to take advantage of multi-core processors.

Receive side scaling (RSS) is a hardware technology for multiple queue support in hardware NIC. Essentially, an RSS-capable NIC load balances incoming packets across different hardware queues at the connection level [13], and one CPU is dedicated to one queue for packet processing in software. Therefore, RSS enables multiple CPUs to process packets in parallel, by considering cache locality.

In this subsection, we start with the discussion of the multithread backend driver in Subsection 4.3.1 and then introduce the concept of RSS to the backend driver to effectively leverage multi-core processors in Subsection 4.3.2.

4.3.1 Multithread Backend Driver

There are multiple solutions to implement a multithread backend driver: multiple delayed interrupts (tasklets in Linux), multiple kernel threads, and multiple high priority kernel threads. The multiple tasklets solution reuses the existing Xen backend driver framework. It runs the backend driver in the context of an interrupted CPU and can achieve RSS, by carefully configuring the interrupt delivery to virtual CPU of domain 0. However, it may lead to long time delays in executing an interrupted thread, and it sacrifices system responsiveness.

The multiple kernel threads solution brings high flexibility in implementation and configuration to leverage OS scheduling policy to achieve both performance and responsiveness. But it may degrade throughput due to the OS scheduling overhead and pre-emption. This solution provides a trade-off between network performance and system responsiveness.

The multiple high priority kernel threads solution sits between them. It uses high priority kernel tasks (and

hence achieves high flexibility in implementation and configuration, leveraging OS scheduling policy) to process network packets in the backend driver, and disables the pre-emption till processing is completed. As a result, it may lead to long time delays in executing other threads, as well.

4.3.2 Receive Side Scaling

Based on the parallelized backend driver, we introduce the concept of RSS to efficiently load balance workloads across CPUs. The virtual interrupts are dynamically load balanced among virtual processors, by forwarding the different physical interrupts to different virtual processors, to achieve virtual RSS using the hardware RSS. In the multiple tasklet solution, CPUs receive interrupts and process the packets from dedicated queues or functions (distributed across queues based on connections) in parallel. In the multiple kernel threads or high priority threads solution, each thread of the backend driver can be bound to a specific virtual CPU, so that the interrupt from NIC can interrupt the corresponding VCPU and schedule threads to process network packets from dedicated queues or functions in time.

We implemented all three approaches and evaluated their performance. Our experimental results (see Section 5) point out that the multiple tasklet solution achieves the best performance.

5. Performance Evaluation

5.1 Experiment Configuration

The 'server' system is an Intel Xeon 5560 platform, equipped with two quad-core processors with SMT-enabled (16 threads in total), running at 2.8 GHZ, with 24 GB 3-channel DDR3 memory. We use ten port (two 4-port and one 2-port) Gigabit Intel 82576 NICs to measure performance in both 1 Gbps and 10 Gbps environments. The performance of virtual interrupt coalescing and the adaptive multi-layer interrupt coalescing is measured with a 1 Gbps network, while the performance of virtual receive side scaling is measured with an aggregated 10 Gbps network, to stress the CPU utilization.

The server uses the Xen 3.4 64 bit version hypervisor. Both domain 0 and guest VM run RHEL5U3. The guest VM runs in a hardware virtual machine (HVM), with a paravirtualized network driver. Netperf UDP_STREAM & TCP_STREAM benchmarks are run as servers in the guest VM, to measure the receive side performance. Unless explicitly mentioned, we used 1472-byte packet size and default 4 KHz interrupt frequency in Intel 1 GB drivers.

In the experiments of virtual interrupt coalescing and adaptive multi-layer interrupt coalescing, guest VMs (up to 9) run on the same socket with domain 0. In the virtual receive side scaling experiment, domain 0 employs 8 VCPUs and binds each of them to a thread in different cores, and a guest VM runs with only one VCPU bounded evenly to the rest of the threads. The backend driver evenly distributes the network request from guest VMs to the 10 GbE NIC per the guest number, that is guest 1, 11, 21, 31,41,51 and 61 share the 1st NIC, etc.

The 'client' has the same hardware configuration as the 'server', but runs 64 bit RHEL5U3. A netperf with 2 threads per GbE connection is run to avoid CPU competition.

5.2 Virtual Interrupt Coalescing

Figure 6-9 show the performance results of netperf UDP-Rx, TCP-Rx with virtual interrupt coalescing of 1KHz. In the UDP-Rx test case with small packets (shown in Figure 6), the throughput drops slightly with virtual interrupt coalescing. That is because the number of arrival packets is larger than the number the software stack (virtual switch and backend driver, etc.) can process. Therefore, many packets are dropped. Virtual interrupt coalescing makes this situation worse. In contrast to UDP, TCP processing with virtual interrupt coalescing, shown in Figure 8, demonstrates 12.6% CPU utilization savings and 3% throughput improvement with small packets. That is mainly because TCP is a reliable in-order protocol and avoids packet loss. When we come to large packets, virtual interrupt coalescing is a performance advantage, in terms of CPU utilization, for both UDP and TCP cases. As shown in Figure 7 and Figure 9, it reduces CPU utilization 3.3% and 14.5% for UDP and TCP, respectively. In addition, we also found from these figures that the frontend driver virtual interrupt coalescing achieves slightly better performance than virtual interrupt coalescing in the backend driver, mainly due to less context switches.

In addition, we also study the performance benefits of virtual interrupt coalescing, as the number of VMs increases. Experimental results are presented in Figures 10 and 11. We observe from the two figures that the benefits of virtual interrupt coalescing increases as VM# goes up. When the number of VMs reaches up to 9, virtual interrupt coalescing in the TCP case largely reduces CPU utilization from 200% to 129%, saving up to 71%.

Frontend interrupt coalescing demonstrates better performance than backend interrupt coalescing, with more CPU utilization savings, due to the reduced context switch between guest and domain 0.

Meanwhile, TCP shows larger performance improvement than UDP among all different configurations. This is due to the larger improvement of cache pollution in the TCP case. In the case of TCP, both domain 0 and guest VM need to access both receive and transmission virtual descriptor queues (that is, to map the memory, access the virtual ring, and copy the packets), thus having more cache pollution than UDP, which only processes received packets.

In summary, we can draw two conclusions: 1) virtual interrupt coalescing can reduce CPU utilization by up to 71%, and 2) frontend driver coalescing achieves better performance than backend coalescing, due to fewer context switches between VMs.

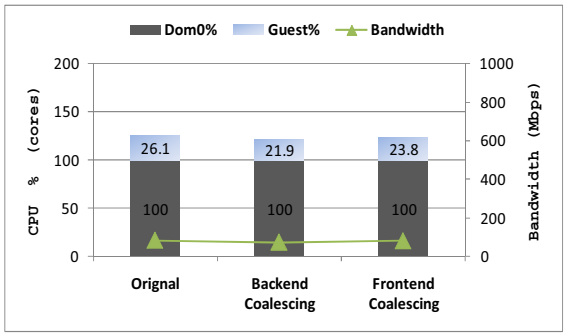


Figure 6. UDP Performance with 50-byte packets

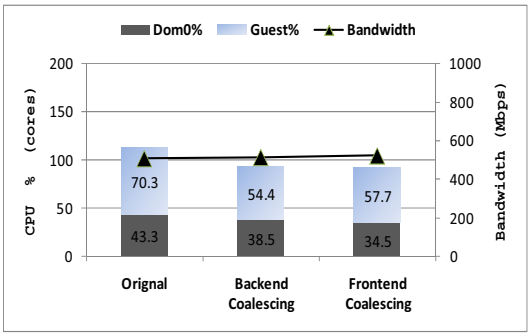


Figure 8. TCP Performance with 50-byte packets

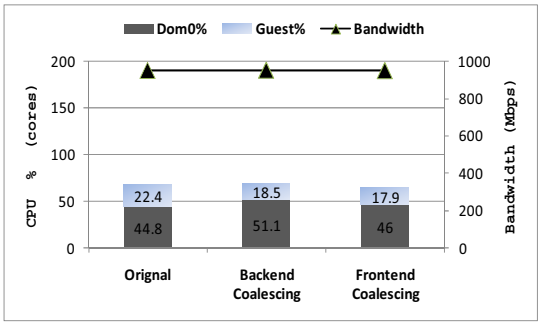


Figure 7. UDP Performance with 1472-byte packets

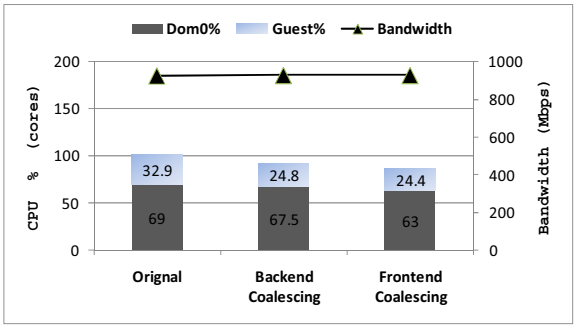


Figure 9. TCP Performance with 1472-byte packets

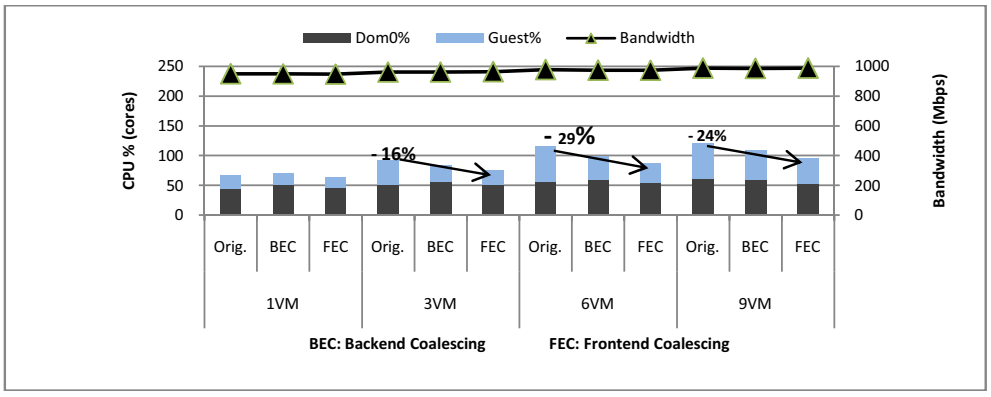


Figure 10: UDP Performance with Multiple VMs

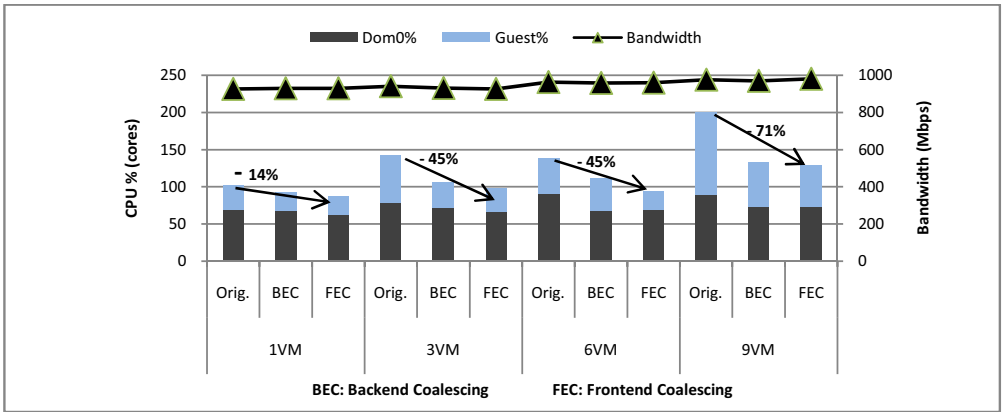


Figure 11. TCP-Rx Performance with Multiple VMs

5.3 Adaptive Multi-Layer Interrupt Coalescing

Going further, we also conduct experiments to understand the performance of our adaptive multi-layer interrupt coalescing. Figure 12-15 illustrate the TCP-Rx performance results of multi-layer interrupt coalescing with 1 guest VM, 3 guest VMs, 6 guest VMs, and 9 guest VMs, respectively. Different latency allocation policies, in the step of 250 *us*, are used between domain 0 and guest VMs, with fixed total latency of 1250 *us*. Among all the configurations, the bandwidth remains constant.

As shown in Figure 12, the best performance latency allocation in the single guest case happens when domain 0 is configured with 500 *us* latency, and the guest is configured with 750 *us* latency. The second best configuration appears in the case when domain 0 is configured with 750 *us* latency, indicating that the best configuration of domain 0 latency is close to 500 *us*, but somewhere between 500 *us* and 750 *us*.

In the 3-guest case, shown in Figure 13, although the best situation happens when domain 0 is configured with 500 *us* latency as 1-guest case, the second best configuration happens when domain 0 is configured with 250 *us* latency, indicating that the best configuration of domain 0 latency is close to 500 *us* but somewhere between 500 *us* and 250 *us*, complying with the trend of Equation (4), when the number of guests increases.

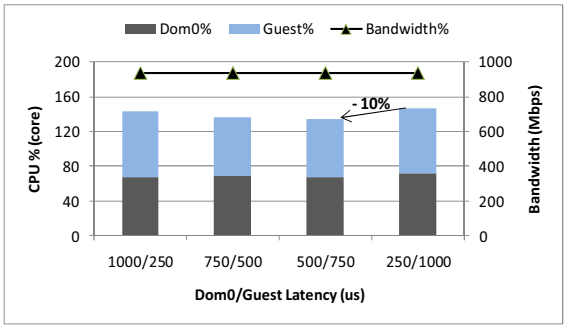


Figure 12. TCP Performance of Multi-Layer Interrupt Coalescing with 1 Guest

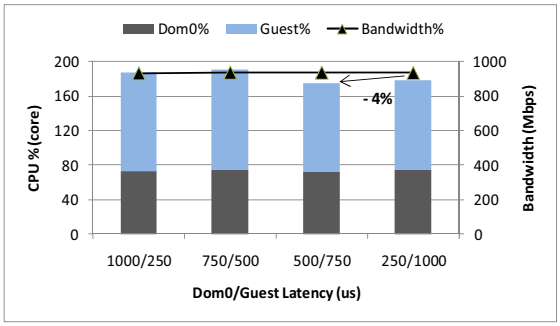


Figure 13. TCP Performance of Multi-Layer Interrupt Coalescing with 3 Guests

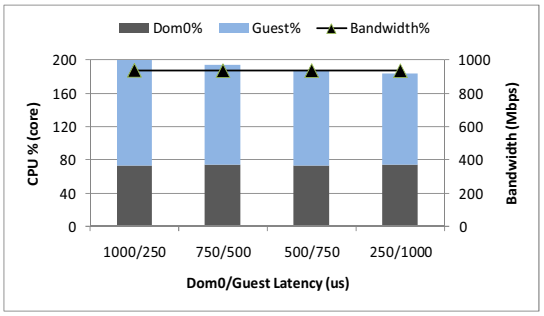


Figure 14. TCP Performance of Multi-Layer Interrupt Coalescing with 6 Guests

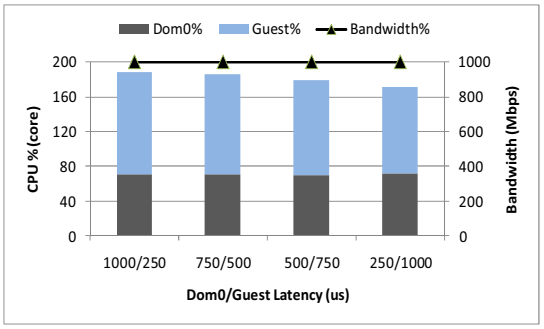


Figure 15. TCP Performance of Multi-Layer Interrupt Coalescing with 9 Guests

The 6-guest and 9-guest configurations, shown in Figure 14 and Figure 15, further reveal the trend of Equation (4). When more guests are used, the best performance configuration moves toward the situation where guest VM has lower interrupt frequency, that is 250us latency for domain 0, and 1000us for guest VM, to save more CPU cycles spent in guest VMs, which is multiplied by the number of guests.

5.4 Virtual Receive Side Scaling

In this subsection, we evaluate the performance benefits of our virtual receive side scaling. In experiments, we run 10 VMs connecting with 10 client threads through aggregated 10 GbE NIC in domain 0. The result is illustrated in Figure 16. We observed in the figure that the multiple kernel threads approach achieves 2.18x performance of the original solution, with an additional 645% CPU utilization. The multiple high priority kernel thread solution obtains a 2.61x throughput of the original solution, reaching 9.5 Gbps line rate with an additional 581% CPU utilization, an additional 0.43x throughput improvement and 64% CPU utilization saving. Multiple tasklet solution has the best performance, achieving 9.5 Gbps line rate with only a 502% CPU utilization. It saves 79% CPU utilization over multiple high priority kernel threads because kernel threads introduce the heavy OS scheduling overhead.

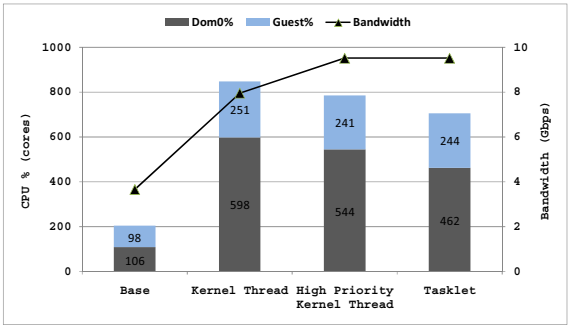


Figure 16. TCP Performance of Virtual Receive Side Scaling

In summary, the virtual receive side scaling can achieve 2.61x throughput over the baseline in the aggregated 10 GbE environment. The multiple tasklets solution obtains the best performance and has a 0.43x performance advantage over the multiple kernel thread solution (from 2.18x to 2.61x).

6. Related Work

New usage models, based on virtualization, have been well explored, to take the advantage of the new layer of software remapping. Hypervisor-based fault tolerance is achieved through coordinating a primary virtual machine with its backup [3]. SMP-Revirt replays execution for multiple-processor VM [8]. Intrusion analysis is enabled through VM logging and replay [11]. LiteGreen saves energy in networked desktops using virtualization [6]. Live migration of a virtual machine through full system trace and replay [14][17] and asynchronous VM replication [5] are studied. They take advantage of virtual I/O, which is underlying hardware neutral. None of them fully solve the virtual network I/O performance issue.

In virtualization, hardware-assisted solutions are proposed to achieve high-performance, such as VMDq [26] and its optimization [24], self-virtualized devices [23], VMM-bypass I/O [18], direct I/O [7], and SR-IOV [8]. In these solutions, each VM is assigned a portion of hardware resource for dedicated access to reduce the hypervisor intervention to the bulk data path. However, they all suffer from hardware stickiness, and thus sacrifice guest transparent services, such as VM based replication and checkpoint, etc.

Characterization and optimization of software network virtualization are well studied. Menon improved Xen network virtualization with redefined interface to incorporate high-level network offload features and optimized data transfer path [22]. Liu accelerates I/O virtualization using dedicated CPU cores [19]. Liao improved network virtualization performance with cache-aware scheduler, virtual switch enhancement, and transmit queue length optimization [20]. Guo analyzed the overhead of network virtualization in 10 GbE and improved performance with enhanced cache affinity in interrupt delivery and hypervisor scheduler [12]. Kesavan investigated I/O service differentiation and performance isolation for virtual machines on individual multi-core nodes in cloud platforms with differential virtual time [15]. They didn't study the efficient interrupt coalescing and virtual receive side scaling.

Interrupt coalescing [29] and load balance among different CPUs, such as receive side scaling [13], are implemented in the native networking environment to improve the efficiency of CPU processing to network packets, however they are not addressed in virtual environment and multiple layer interrupt coalescing.

Virtual interrupt coalescing is explored in disk I/O [1], adaptive interrupt coalescing is employed in the virtual function of an SR-IOV-capable device [8]. They didn't address the network latency and multiple layer interrupt coalescing issues.

7. Conclusion

In this paper, we first examined Xen network I/O virtualization challenges and then proposed optimizations to reduce the I/O virtualization overhead: efficient interrupt coalescing for virtualization and virtual receive side scaling. We implemented our proposed optimizations in Xen and conducted detailed experiments to understand their benefits. Our experimental results reveal that our efficient interrupt coalescing for virtualization can significantly reduce CPU utilization, and virtual receive side scaling is able to obtain 2.2x bandwidth over the baseline in the 10 Gbps network environment.

Acknowledgment

The other members contributing to this study include Xiaowei Yang, Xiantao Zhang, and the Intel Linux OS VMM team. We would like to thank Nakajima Jun and Wilfred Yu for their excellent insights and suggestions, as well.

8. References

[1] I. Ahmad, A. Gulati, A. Mashtizadeh, Maxime Austruy, Improving Performance with Interrupt Coalescing for Virtual Machine Disk IO in VMware ESX Server, http://www.vmware.com

[2] P. Barham, B. Dragovic, K. Fraser et al., Xen and the art of virtualization, 19th ACM symposium on Operating Systems Principles (SOSP), NY, 2003.

[3] T. C. Bressoud, F. B. Schneider, Hypervisor-based fault tolerance, ACM Transactions on Computer Systems (TOCS), 1996.

[4] C. Clark, K. Fraser, S. Hand et al., Live migration of virtual machines, 2nd Symposium on Networked Systems Design & Implementation (NSDI), Boston, MA, 2005.

[5] B. Cully, G. Lefebvre, D. Meyer et al., Remus: High Availability via Asynchronous Virtual Machine Replication, 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI), San Francisco, CA, 2008.

[6] T. Das, P. Padala et al., LiteGreen: saving energy in networked desktops using virtualization, USENIX Annual Technical Conference, Boston, MA, 2010.

[7] Y. Dong, J. Dai, et al. Towards high-quality I/O virtualization. Israeli Experimental Systems Conference (SYSTOR), Haifa, Israel, 2009.

[8] Y. Dong, X. Yang, X. Li et al., High Performance Network Virtualization with SR-IOV, 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA), Bangalore, India, 2010.

[9] Y. Dong, Improving Virtualization Performance and Scalability with Advanced Hardware Accelerations, 2010 IEEE International Symposium on Workload Characterization, Atlanta, GA, 2010.

[10] G. W. Dunlap, S. T. King, S. Cinar et al., Revirt: Enabling intrusion analysis through virtual machine logging and replay, 5th Symposium on Operating Systems Design and Implementation (OSDI), Boston, MA, 2002.

[11] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman et al., Execution replay of multiprocessor virtual machines, 2008 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), Seattle, WA, 2008.

[12] D. Guo, G. Liao, L. N. Bhuyan: Performance characterization and cache-aware core scheduling in a virtualized multi-core server under 10GbE, 2009 IEEE International Symposium on Workload Characterization (IISWC), Austin, TX, 2009.

[13] Intel Corporation, Intel® 82576 Gigabit Ethernet Controller Datasheet. http://www.intel.com

[14] A. Kadav and M. M. Swift, Live Migration of Direct-Access Devices, ACM SIGOPS Operating Systems Review (OSR), Volume 43 Issue 3, July 2009.

[15] M. Kesavan, A. Gavrilovska et al., Differential Virtual Time (DVT): Rethinking I/O Service Differentiation for Virtual Machines, 1st ACM symposium on Cloud computing (SOCC), Indianapolis, IN, 2010.

[16] Kernel Based Virtual Machine, http://www.linux-kvm.org/

[17] H. Liu, H. Jin et al., Live migration of virtual machine based on full system trace and replay, 18th ACM international symposium on High performance distributed computing (HPDC), Munich, Germany, 2009.

[18] J. Liu, et al. High Performance VMM-Bypass I/O in Virtual Machines, USENIX Annual Technical Conference, Boston, MA, 2006.

[19] J. Liu , B. Abali, Virtualization Polling Engine (VPE): Using Dedicated CPU Cores to Accelerate I/O Virtualization, 23rd international conference on Supercomputing, Yorktown Heights, NY, 2009.

[20] G. Liao, D. Guo, L. N. Bhuyan et al., Software Techniques to Improve Virtualized I/O Performance on Multi-core Systems, ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), San Jose, CA, 2008.

[21] M. Lee, A. S. Krishnakumar, P. Krishnan et al., Supporting soft real-time tasks in the xen hypervisor, 2010 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), Pittsburgh, PA, 2010.

[22] A. Menon, A. L. Cox, W. Zwaenepoel, Optimizing Network Virtualization in Xen, USENIX Annual Technical Conference, Boston, MA, 2006.

[23] H. Raj, K. Schwan, High Performance and Scalable I/O Virtualization via Self-Virtualized Devices, 16th IEEE International Symposium on High Performance Distributed Computing (HPDC), Monterey Bay, CA, 2007.

[24] K. Ram , J. R. Santos et al., Achieving 10 Gb/s Using Safe and Transparent Network Interface Virtualization, 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE), Washington, DC, 2009.

[25] J. Salim, When NAPI Comes to Town, Proceedings of Linux 2005 Conference, UK, August 2005.

[26] J. R. Santos, Y. Turner, G. Janakiraman et al., Bridging the Gap between Software and Hardware Techniques for I/O Virtualization, 2008 USENIX Annual Technical Conference, Boston, MA, 2008.

[27] J. Sugerman, G. Venkitachalam, B. Lim, Virtualizing I/O devices on VMware Workstation's Hosted Virtual Machine Monitor, 2002 USENIX Annual Technical Conference, Boston, MA, 2001.

[28] M. Ben-Yehuda etc., The Turtles Project: Design and Implementation of Nested Virtualization, 9th Symposium on Operating Systems Design and Implementation (OSDI), Vancouver, BC, Canada, 2010.

[29] M. Zec, M. Mikuc, M. Zagar, Integrated performance evaluating criterion for selecting between interrupt coalescing and normal interruption, International Journal of High Performance Computing and Networking, 2005.