Is I/O virtualization ready for End-to-End Application Performance?

J. Lakshmi, S. K. Nandy
Indian Institute of Science, Bangalore, India
{jlakshmi,nandy}@serc.iisc.ernet.in

Abstract: *Workload consolidation using system virtualization feature is the key to many successful green initiatives in data centres. In order to exploit the available compute power, such systems warrant sharing of other hardware resources like memory, caches, I/O devices and their associated access paths among multiple threads of independent workloads. This mandates the need for ensuring end-to-end application performance. In this paper we explore the current practices for I/O virtualization, using sharing of the network interface card as the example, with the aim to study the support for end-to-end application performance guarantees. To ensure end-to-end application performance and limit interference caused due to the sharing of devices, we present an evaluation of previously proposed end-to-end I/O virtualization architecture. The architecture is an extension to the PCI-SIGV specification of I/O hardware to support reconfigurable device partitions and uses VMM-bypass technique for device access by the virtual machines. Simulation results of the architecture for application quality of service guarantees demonstrate the flexibility and scalability of the architecture.*

Keywords – Multicore server virtualization, IO-virtualization architectures,QoS, Performance.

I. Introduction

Multi-core servers have brought in tremendous computing capacity to the commodity systems. These multi-core servers have not only prompted applications to use fine-grained parallelism to gain advantage of the abundance of CPU cycles, they have also initiated the coalescing of multiple independent workloads onto a single server. Multicore servers combined with system virtualization have led to many successful green initiatives of data centre workload consolidation. This consolidation however needs to satisfy end-to-end application performance guarantees. Current virtualization technologies have evolved from the prevalent single-hardware single-OS model which presumes the availability of all other hardware resources to the current scheduled process. This causes performance interference among multiple independent workloads sharing an I/O device, based on the individual workloads. Major efforts towards consolidation have focussed on aggregating the CPU cycle requirement of the target workloads. But I/O handling of these workloads on the consolidated servers results in sharing of the physical resources and their associated access paths. This sharing causes interference that is dependent on the consolidated workloads and makes the application performance non-deterministic [1][2][3]. In such scenarios, it is essential to have appropriate mechanisms to define, monitor and ensure resource sharing policies across the various contending workloads. Many applications like real-time hybrid voice and data communication systems onboard aircraft and naval vessels, streaming and on-demand video delivery, database and web-based services, when consolidated onto virtualized servers, need to support soft real-time application deadlines to ensure performance.

Standard I/O devices are not virtualization aware and hence their virtualization is achieved using a software layer for multiplexing device access to independent VMs. In such cases I/O device virtualization is commonly achieved following two basic modes of virtualization, namely para-virtualization and emulation [4]. In para-virtualization the physical device is accessed and controlled using a protected domain which could be the virtual machine monitor (VMM) itself or an independent virtual machine (VM), also called the independent driver domain (IDD) as in Xen. The VMM or IDD actually do the data transfer to and from the device into their I/O address space using the device's native driver. From there the copy or transfer of the data to the VM's address space is done using what is commonly called the para-virtualized device driver. The para-virtualized driver is specifically written to support a specific mechanism of data transfer between the VMM/IDD and the VM and needs a change in the OS of the VM (also called GuestOS).

In emulation, the GuestOS of VM installs a device driver of the emulated virtual device. All the calls of this emulated device driver are trapped by the VMM and translated to the native device driver's calls. The advantage of the emulation is that it allows the GuestOS to be unmodified and hence easier to adopt. However, para-virtualization has been found to be much better in performance when compared to emulation. This is because emulation results in each instruction translation whereas para-virtualization involves only page-address translation. But, both these modes of device virtualization impose resource overheads when

compared to non-virtualized servers. These overheads translate into application performance loss.

The second drawback of the existing architectures is their lack of sufficient quality of Service (QoS) controls to manage device usage on a per VM basis. A desirable feature of these controls is that they should guarantee application performance with specified QoS on the shared device and this performance should be unaffected by the workloads sharing the device. The other desirable feature is that the unused device capacity should be available for use to the other VMs. Prevalent virtualization technologies like *Xen and Vmware* and even standard *linux* distributions use a software layer within the network stack to implement NIC usage policies. Since these systems were built with the assumption of single-hardware single-OS model, these features provide required control on the outgoing traffic from the NIC of the server. The issue is with the incoming traffic. Since the policy management is done above the physical layer, ingress traffic accepted by the device is later dropped based on input stream policies. This results in the respective application not receiving the data, which perhaps satisfies the application QoS, but causes wasted use of the device bandwidth that affects the delivered performance of all the applications sharing the device. Also, it leads to non-deterministic performance that varies with the type of applications using the device. This model is insufficient for the virtualized server supporting sharing of the NIC across multiple VMs. In this paper we describe and evaluate an end-to-end I/O virtualization architecture that addresses these drawbacks.

Rest of the paper is organized as follows. section II presents experimental results on existing virtualization technologies, namely Xen and Vmware, that motivate this work; section III then describes an end-to-end I/O virtualization architecture to overcome the issues raised in section II; section IV details the evaluation of the architecture and presents the results; section V highlights the contributions of this work with respect to existing literature and section VI details the conclusions.

II. Motivation

Existing I/O Virtualization architectures use extra CPU cycles to fulfill equivalent I/O workload. These overheads reflect in the achievable application performance, as depicted by the graph of Figure 1. The data in this graph represents achievable throughput by the *httperf* [5] benchmark hosted on a non-virtualized and virtualized Xen[6] and Vmware-ESXi [7] servers. In each of the case the *http* server was hosted on a *Linux(FC6)* OS and for the virtualized server, the hypervisor, IDD (*Xen*) [8] and the virtual machine were pinned to use the same physical CPU. The server used was dual core Intel Core2Duo system with 2GB RAM and 10/100/1000Mbps NIC. In the *Xen* hypervisor the virtual NIC used by the VM was configured to use a para-virtualized device driver implemented using event channel mechanism and a software bridge for creating virtual NICs. In the case of *Vmware* hypervisor the virtual NIC used inside the VM was configured using a software switch with access to device through emulation.

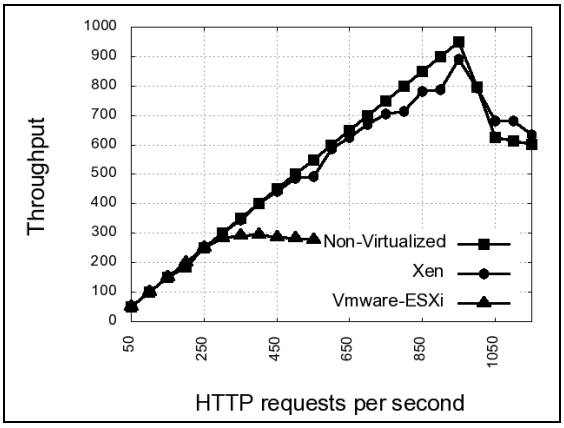

Figure 1: *httperf* benchmark throughput graph for non-virtualized, Xen and Vmware-ESXi virtual machine hosting http server on Linux(FC6). [1]

As can be observed from the graphs of Figure 1, the sustainable throughput of the benchmark drops considerably when the *http* server is moved to a virtualized server when compared to the non-virtualized server. The reason for this drop is answered by the CPU Utilization graph depicted in Figure 2. From the graphs we notice that moving the http server from non-virtualized to virtualized server, the %CPU utilization, to support the same *httperf* workload, is increased significantly and this increase is substantial for the emulated mode of device virtualization. The reason for this increased CPU utilization is because of I/O device virtualization overheads.

Further, when the same server is consolidated with two VMs sharing the same NIC, each supporting one stream of an independent *httperf* benchmark, there is further drop of achievable throughput per VM. This is explicable since each VM now contends for the same NIC. The virtualization mechanisms not only share the device but also the device access paths. This sharing causes serialization which leads to latencies and application performance loss which is dependent on

[1] Some data of the graph has been reused from [9][10]

the nature of the consolidated workloads. Also, the increased latencies in supporting the same I/O workload on the virtualized platform causes loss of usable device bandwidth which further reduces the scalability of device sharing by multiple VMs.

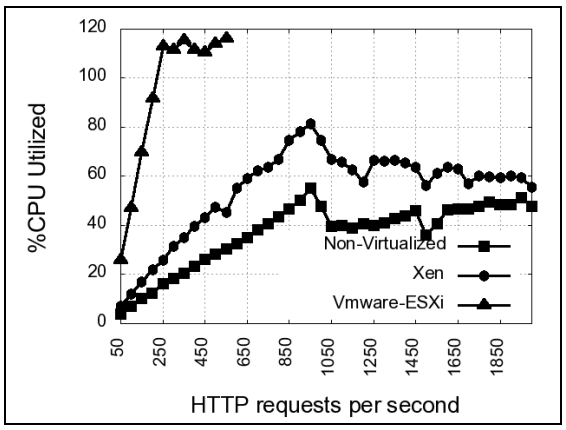

Figure 2: CPU resource utilized by the http server to support httperf benchmark throughput.

This scalability can be improved to some extent by pinning different VMs to independent cores and using a high speed, high bandwidth NIC. Still the high virtualization overheads coupled with serialization due to shared access paths restricts device sharing scalability.

The next study is on evaluating the NIC specific QoS controls existing in Xen and Vmware. Since current NICs do not support QoS controls these are provided by the OS managing the device. In either case, these controls are implemented in the network stack above the physical device. Because of this they are insufficient as is displayed by the graphs in Figure 3and Figure **4**.

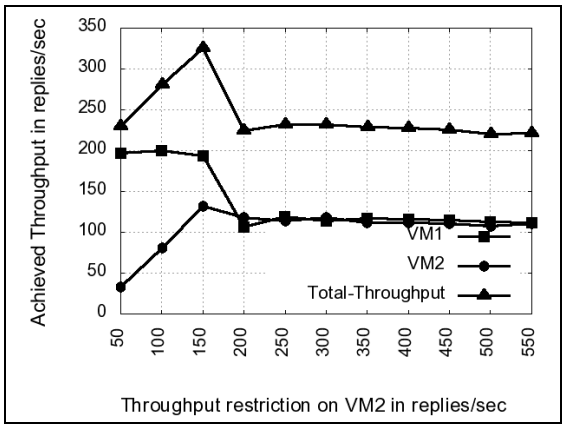

Figure 3: *httperf* achievable throughput on a Vmware-ESXi consolidated server with NIC sharing and QoS guarantees.

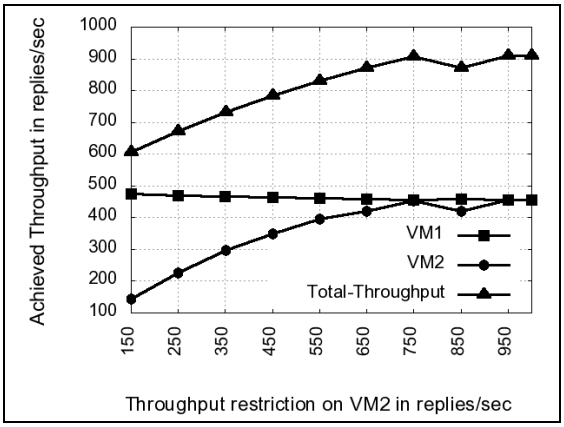

Figure 4: *httperf* achievable throughput on a Xen consolidated server with NIC sharing and QoS guarantees.

Each of the graphs shows the maximum achievable throughput by VM1 when VM2 is constrained by a specified QoS guarantee. This guarantee is the maximum throughput that VM2 should deliver and is implemented using the network QoS controls available in *Xen* and *Vmware* servers. In *Xen* these QoS controls are implemented using *tc* utilities of the *netfilter* module of *linux* OS of the *Xen-IDD.* In *Vmware*, *Veam* enabled network controls are used. VM1 and VM2 are two different VMs hosted on the same server sharing the same NIC. We observe that for the unconstrained VM, in this case VM1, maximum achievable throughput does not exceed 200 for *Vmware* and 475 for *Xen*. This is considerably low when compared to the maximum achievable throughput for a single VM using the NIC. The reason being, the constrained VM, namely VM2, is receiving all requests. VM2 is also processing these requests and generating appropriate replies which results in CPU resource consumption. Only some replies of the received requests are dropped based on the currently applicable QoS on the usable bandwidth. This is because both *Vmware* and *Xen* support QoS controls on the egress traffic at the NIC. This approach of QoS control on resource usage is wasteful and coarse-grained. As can be observed, as the constraint on VM2 is relaxed the behaviour of NIC sharing reaches best effort and the resulting throughput achievable by any of the VM is obviously less than what can be achieved when a single VM is hosted on the server. These graphs clearly demonstrate the insufficiency of the existing QoS controls.

From the above experiments we conclude the following drawbacks in the existing I/O virtualization architectures:

- Building hypervisors or VMMs using single-hardware single-OS model leads to cohesive architectures leading to high virtualization overheads. Virtualization overheads being high cause loss of usable device bandwidth. This

often results in under-utilized resources and limited consolidation ratios, particularly for I/O workloads. The remedy to this approach is to build I/O devices that are virtualization aware and decouple the device management from device access i.e, provide native access to the I/O device from within the VM and allow VMM to manage concurrency issues rather than the ownership issues.

- Lack of fine-grained QoS controls on device sharing cause performance loss which is dependent on the workloads of the VMs sharing the device. This leads scalability issues in sharing the I/O device. To address this the I/O device should support QoS controls for both the incoming and outgoing traffic.

To overcome the above drawbacks, we propose an I/O virtualization architecture. This architecture proposes an extension to the PCI-SIG IOV specification [11] for virtualization enabled hardware I/O devices with a VMM-bypass [12] mechanism for virtual device access.

III. End-to-End I/O Virtualization Architecture

We propose an end-to-end I/O virtualization architecture that enables direct or native access to the I/O device from within the VM rather than accessing it through the layer of VMM or IDD. PCI-SIG IOV specification proposes virtualized I/O devices that can support native device access by the VM, provided the hypervisor is built to support such architectures. IOV specified hardware can support multiple virtual devices at the hardware level. The VMM needs to be built such that it can recognize and export each virtual device to an independent VM, as if the virtual device was an independent physical device. This allows native device access to the VM. When a packet hits the hardware virtualized NIC, the VMM should recognize the destination VM of an incoming packet by the interrupt raised by the device and forwards it to the appropriate VM. The VM processes the packet as it would do so in the case of non-virtualized environment. Here, device access and scheduling of device communication is managed by the VM that is using the device. This eliminates the intermediary VMM/IDD on the device access path and reduces I/O service time which improves the usable device bandwidth and application throughput.

To support the idea of QoS based on device usage, we extend the IOV architecture specification by enabling reconfigurable memory on the I/O device. For each of the virtual device defined on the physical device, the device memory associated with the virtual device is derived from the QoS requirement of the VM to which the virtual device is allocated. This, along-with features like TCP offload, virtual device priority and bandwidth specification support at the device level provide fine-grained QoS controls at the device while sharing it with other VMs, as is elaborated upon in the evaluation section.

Figure 5 gives a block schematic of the proposed I/O virtualization architecture.[2] The picture depicts a NIC card that can be housed within a virtualized server. The card has a controller that manages the DMA transfer to and from the device memory. The standard device memory is replaced by a re-partitionable memory supported with n sets of device registers. A set of m memory partitions, where $m \leq n$, with device registers forms the virtual-NICs (vNICs). Ideally the device memory should be reconfigurable, i.e. dynamically partitionable, and the VM's QoS requirements would drive the sizing of the memory partition. The advantage of having a dynamically partitionable device memory is that any unused memory can be easily extended into or reduced from a vNIC in order to match adaptive QoS specifications. The NIC identifies a vNIC request by generating message signaled interrupts (MSI). The number of interrupts supported by the controller restricts the number of vNICs that can be exported. Based on the QoS guarantees a VM needs to honour, judicious use of native and para-virtualized access to the vNICs can overcome this limitation. A VM that has to support stringent QoS guarantees can choose to use native access to the vNIC whereas those VMs that are looking for best-effort NIC access can be allowed para-virtualized access to the vNIC.

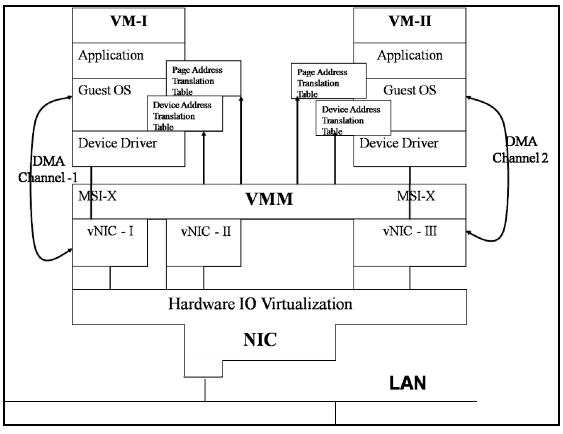

Figure 5: NIC architecture supporting MSI interrupts with partitionable device memory, multiple device register sets and DMA channels enabling independent virtual-NICs.

[2] This section is being reproduced from [10] to maintain continuity in the text. Complete architecture description with performance statistics on achievable application throughput can be found in [10].

The VMM can aid in setting up the appropriate hosting connections based on the requested QoS requirements.

The proposed architecture can be realized by the following modifications:

- **Virtual-NIC:** In order to define vNIC, the physical device should support time-sharing in hardware. For a NIC this can be achieved by using MSI and dynamically partitionable device memory. These form the basic constructs to define a virtual device on a physical device as depicted in Figure 5. Each virtual device has a specific logical device address, like the MAC address in case of NICs, based on which the MSI is routed. Dedicated DMA channels, a specific set of device registers and a partition of the device memory are part of the virtual device interface which is exported to a VM when it is started. We call this virtual interface as the virtual-NIC or vNIC which forms a restricted address space on the device for the VM to use and remains in possession of the VM till it is active or relinquishes the device.

- **Accessing virtual-NIC:** For accessing the virtual-NIC native device driver is hosted inside the VM and is initialized with the help of VMM when the VM is initialized. This device driver can only manipulate the restricted device address space which was exported through the vNIC interface by the VMM. With the vNIC, the VMM only identifies and forwards the device interrupts to the destination VM. The OS of the VM now handles the I/O access and thus can be accounted for the resource usage it incurs. This eliminates the performance interference due to VMM/IDD handling multiple VMs' request to/from a shared device. Also, because the I/O access is now directly done by the VM, the service time on the I/O access reduces thereby resulting in better bandwidth utilization. With the vNIC interface, data transfer is handled by the VM. While initializing the device driver for the virtual NIC the VM sets up the Rx/Tx descriptor rings within its address space and makes request to the VMM for initializing the I/O page translation table. The device driver uses this table and performs DMA transfers directly into the VM's address space.

- **QoS and virtual-NIC:** The device memory partition acts as a dedicated device buffer for each of the VMs and with appropriate logic on the NIC card one can easily implement QoS based SLAs on the device that translate to bandwidth restrictions and VM specific priority. The key is being able to identify the incoming packet to the corresponding VM, which the NIC is now expected to do. While communicating, the NIC controller decides on whether to accept or reject the incoming packet based on the bandwidth specification or the virtual device's available memory. This gives a fine-grained control on the incoming traffic and helps reduce the interference effects. The outbound traffic can be controlled by the VM itself using any of the mechanisms as is done in the existing architectures.

IV. Architecture Evaluation for QoS controls

The proposed architecture was generated using Layered Queuing Network (LQN) Model and service times for the various entries of the model were obtained by using runtime profilers on the actual Xen based virtualized server. Complete model building and validation details are available in [9][10]. Here we present the results of QoS evaluation carried out using the LQN model [12] of the proposed architecture. The QoS experiments were conducted along the same lines as described in the introduction section. The difference now is that the QoS control is applied on the ingress traffic of the constrained VM, namely VM2. The results obtained are depicted in Figure 6.The proposed architecture allows for achieving higher application throughput on the shared NIC firstly because of the VMM-bypass [12]. Also, as can be observed from the graphs above, the control of ingress traffic in the case of *httperf* benchmark shows highly improved performance benefit to the unconstrained VM, namely VM1.

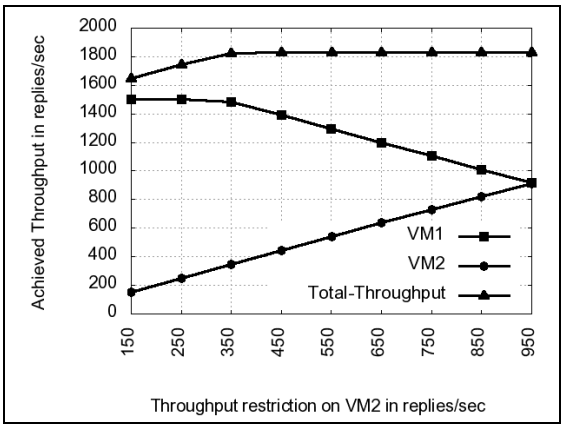

Figure 6: *httperf* throughput sharing on a QoS controlled, shared NIC between two VMs using the proposed architecture with throughput constraints applied on the ingress traffic of VM2 at the NIC.

For request-response kind of benchmarks like the *httperf*, controlling the ingress bandwidth is beneficial because once a request is dropped due to saturation of allocated bandwidth, there is no

downstream activity associated with it and wasteful resource utilization of NIC and CPU is avoided. The QoS control at the device on the input stream of VM2 and the native access to the vNICs by the VMs gives the desired flexibility of making the unused bandwidth available to the unconstrained VM.

V. Related work

In early implementations, I/O virtualization adopted dedicated I/O device assignment to a VM. This later evolved to device sharing across multiple VMs through virtualized software interfaces [14][4]. A dedicated software entity, called the I/O domain is used to perform physical device management. The I/O domain is either part of the VMM or is by itself an independent domain, like the IDD of Xen [8][15]. With this intermediary software layer between the device and the VM, any application in a VM seeking access to the device has to route the request through it. This architecture still builds over the single-hardware single-OS model [16]-[21]. The consequence of such virtualization techniques is visible in the loss of application throughput and usable device bandwidth on virtualized servers as discussed earlier. Because of the poor performance of the I/O virtualization architectures a need to build concurrent access to the shared I/O devices was felt and recent publications on concurrent direct network access (CDNA)[22] [19] and scalable self-virtualizing network interface describe such efforts. However, the scalable self-virtualizing interface [23] describes assigning a specific core for network I/O processing on the virtual interface and exploits multiple cores on embedded network processors for this. The paper does not detail how the address translation issues are handled, particularly in the case of virtualized environments. The CDNA architecture is similar to the proposal in this paper in terms of allowing multiple VM specific Rx and Tx device queues. But CDNA still builds over the VMM/IDD handling the data transfer to and from the device. Although the results of this work are exciting, the architecture still lacks the flexibility required to support fine-grained QoS. And, the paper does not discuss about the performance interference due to uncontrolled data reception by the device nor does it highlight the need for addressing the QoS constraints at the device level. The proposed architecture in this paper addresses these and also the issue of pushing the basic constructs to assign QoS attributes like required bandwidth and priority into the device to get finer control on resource usage and on restricting performance interference.

The proposed architecture has it basis in *exokernel*'s[24] philosophy of separating device management from protection. In *exokernel*, the idea was to extend native device access to applications with *exokernel* providing the protection. In our approach, the extension of native device access is to the VM, the protection being managed by the VMM. A VM is assumed to be running the traditional OS. Further, the PCI-SIG community has realized the need for I/O device virtualization and has come out with the IOV specification to deal with it. The IOV specification however, details about device features to allow native access to virtual device interfaces, through the use of I/O page tables, virtual device identifiers and virtual device specific interrupts. The specification presumes that QoS is a software feature and does not address this. Many implementations adhering to the IOV specification are now being introduced in the market by *Intel*[25], *Neterion*[26], *NetXen*[27], *Solarflare* [28] etc. *CrossBow*[29]suite from *SUN Microsystems* talks about this kind of resource provisioning, but it is a software stack over a standard IOV complaint hardware. The results published using any of these products are exciting in terms of the performance achieved, but almost all of them have ignored the control of reception at the device level. We believe that lack of such a control on highly utilized devices will cause non-deterministic application performance loss and under-utilization of the device bandwidth.

VI. Conclusion

In this paper we described how the lack of virtualization awareness in I/O devices leads to latency overheads on the I/O path. Added to this, the intermixing of device management and data protection issues further increases the latency, thereby reducing the effective usable bandwidth of the device. Also, lack of appropriate device sharing control mechanisms, at the device level leads to loss bandwidth and performance interference on the device sharing VMs. To address these issues we proposed I/O device virtualization architecture, as an extension to the PCI-SIG IOV specification, and demonstrated its benefit through simulation techniques. Results demonstrate that by moving the QoS controls to the shared device, the unused bandwidth is made available to the unconstrained VM, unlike the case in prevalent technologies. The proposed architecture also improves the scalability of VMs sharing the NIC because it eliminates the common software entity which regulates I/O device sharing. The other advantage is that with this architecture, the maximum resource utilization is now accounted for by the VM. Also, this architecture reduces the workload interference on sharing a device and simplifies the consolidation process.

References

[1] M. Welsh and D. Culler, "Virtualization considered harmful: OS design directions for well-conditioned services", Hot Topics in OS, 8th Workshop, 2001.

[2] Kyle J. Nesbit, James E Smith, Miquel Moreto, Francisco J. Cazorla, Alex Ramirez, Mateo Valero, "Multicore Resource Management ", IEEE Micro, Vol. 28, Issues 3, P-6-16, 2008.

[3] Kyle J. Nesbit, Miquel Moreto, Francisco J. Cazorla, Alex Ramirez, Mateo Valero, and James E. Smith, Virtual Private Machines: Hardware/Software Interactions in the Multicore Era, In IEEE Micro special issue on Interaction of Computer Architecture and Operating System in the Manycore Era, May/June 2008.

[4] Scott Rixner, "Breaking the Performance Barrier: Shared I/O in virtualization platforms has come a long way, but performance concerns remain", ACM Queue – Virtualization, Jan/Feb 2008.

[5] D. Mosberger and T. Jin, "httperf: A Tool for Measuring Web Server Performance," ACM, Workshop on Internet Server Performance, pp. 59-67, June 1998.

[6] Paul Barham , Boris Dragovic , Keir Fraser , Steven Hand , Tim Harris , Alex Ho , Rolf Neugebauer , Ian Pratt , Andrew Warfield, "Xen and the art of virtualization", 19th ACM SIGOPS, Oct. 2003.

[7] "VMware ESX Server 2 - Architecture and Performance Implications", 2005, available at http://www.vmware.com/pdf/esx2_performance_implications.pdf

[8] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. War_eld, and M. Williamson, "Safe hardware access with the Xen virtual machine monitor." 1st Workshop on OASIS, Oct 2004.

[9] J.Lakshmi, S.K.Nandy, "Modeling Architecture-OS interactions using Layered Queuing Network Models", International Conference Proceedings of HPC Asia, March, 2009, Taiwan.

[10] J. Lakshmi, S. K. Nandy, "I/O Device virtualization in Multi-core era, a QoS Pespective", Workshop on Grids, Clouds and Virtualization, International Conference on Grids and Pervasive computing, Geneva, May 2009.

[11] PCI-SIG IOV Specification available online at http://www.pcisig.com/specifications/iov

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

[13] Layered Queueing Network Solver software package, http://www.sce.carleton.ca/rads/lqns/

[14] T. von Eicken and W. Vogels. Evolution of the virtual interface architecture. Computer, 31(11), 1998.

[15] J. Sugerman, G. Venkatachalam, and B. Lim. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In Proceedings of the USENIX Annual Technical Conference, June 2001.

[16] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing performance isolation across virtual machines in Xen. In M. van Steen and M. Henning, editors, Middleware, volume 4290 of Lecture Notes in Computer Science, pages 342–362. Springer, 2006.

[17] Weng, C., Wang, Z., Li, M., and Lu, X. 2009. The hybrid scheduling framework for virtual machine systems. In Proceedings of the 2009 ACM SIGPLAN/SIGOPS international Conference on Virtual Execution Environments (Washington, DC, USA, March 11 - 13, 2009).

[18] Kim, H., Lim, H., Jeong, J., Jo, H., and Lee, J. 2009. Task-aware virtual machine scheduling for I/O performance. In Proceedings of the 2009 ACM SIGPLAN/SIGOPS international Conference on Virtual Execution Environments (Washington, DC, USA, March 11 - 13, 2009).

[19] Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the Xen virtual machine environment. In Proceedings of the ACM/USENIX Conference on Virtual Execution Environments, June 2005.

[20] Menon, A. L. Cox, and W. Zwaenepoel. Optimizing network virtualization in Xen. In Proceedings of the USENIX Annual Technical Conference, June 2006.

[21] Santos, J. R., Janakiraman, G., Turner, Y., Pratt, I. 2007. Netchannel 2: Optimizing network performance. Xen Summit Talk (November)

[22] Willmann, P., Shafer, J., Carr, D., Menon, A., Rixner, S., Cox, A. L., Zwaenepoel, W. Concurrent direct network access for virtual machine monitors. In Proceedings of the International Symposium on High-Performance Computer Architecture,2007 (February).

[23] H. Raj and K. Schwan. Implementing a scalable self-virtualizing network interface on a multicore platform. In Workshop on the Interaction between Operating Systems and Computer Architecture, Oct. 2005.

[24] M. Frans Kaashoek, et. Al., "Application Performance and Flexibility on Exokernel Systems ", 16th ACM SOSP, Oct, 1997.

[25] Intel Virtualization Technology for Directed-I/O www.intel.com/technology/itj/2006/v10i3/2-io/7-conclusion.htm

[26] Neterion http://www.neterion.com/

[27] NetXen http://www.netxen.com/

[28] Solarflare Communications http://www.solarflare.com/

[29] CrossBow: Network Virtualization and Resource Control http://www.opensolaris.org/os/community/networking/crossbow_sunlabs_ext.pdf