Quality of Service for I/O Workloads in Multi-core Virtualized Servers

Abstract: *Emerging trend of multi-core servers promises to be the panacea for all data-center issues with system virtualization as the enabling technology. System virtualization gives the ability to create virtual replicas of a complete system, over which independent virtual machines can be built, complete with their own, individual operating systems, software and applications. This provides for complete system isolation of the virtual machines. Apart from the system isolation, one of the key drivers for virtualization adoption in data-centers will be the virtual machine performance isolation that can be achieved over a consolidated server. This chapter exposes the basic requirements on virtualized servers in order to provide performance isolation to virtual machines with a specific focus on enterprise workloads that are a mix of compute and I/O intensive workloads. We evaluate existing, popular system virtualization technologies in terms of the provisions they have for performance isolation and their limitations, and propose an end-to-end system virtualization architecture to overcome these limitations.*

1. Introduction: Multi-core servers coupled with system virtualization are emerging as credible solutions to data-center issues. Today's data-centers have concerns of reducing space and power footprint of its computing infrastructure which the multi-core servers can easily address. A typical multi-core server has enough computing capacity to host several of server applications on a single machine. The most significant issue with co-hosting multiple server applications on a single machine is with the software environment of each of the applications. In fact, non-compliance of software amongst multiple applications is the biggest challenge for server consolidation and virtualization directly addresses this problem. As a matter of fact, server consolidation is the genesis of system virtualization so as to provide software isolation for individual applications. System virtualization gives the ability to create virtual replicas of a complete system, over which independent virtual machines (VMs) can be built, complete with their own, individual operating systems, software and applications. This provides for complete software isolation of the VMs. Apart from the software isolation, one of the key drivers for virtualization adoption in data-centers will be the virtual machine performance isolation that can be achieved over a consolidated server. This is essential particularly for enterprise application workloads, like database, mail, and web-based applications, which have both CPU and I/O workload components. Current commodity multi-core technologies have micro-architecture support for virtualization that provides CPU workload isolation.

In multi-core servers the ratio of cpu-cores to that of I/O interfaces tends to be high. Because of this, the I/O device sharing dynamics change when compared to dedicated servers, wherein all the resources like the processors, memory, I/O interfaces for disk and network access are used predominantly by the hosted application. Hence, solutions that optimize or maximize the application usage of the system resources are sufficient to address the performance of the application. This scenario changes completely when multiple applications are consolidated on to a single, physical, multi-core server. Because of the restricted number of I/O interfaces that can be provisioned on the multi-core servers, many workloads have to share an I/O interface. The challenge is in ensuring performance of the independent I/O intensive applications on the consolidated server while sharing a single I/O device interface across many workloads [1]. A logical solution to this is to provide faster and high-bandwidth I/O interfaces, which happens to be the trend in the current virtualized servers. Despite this, such systems suffer from two distinct problems with regard to current I/O device virtualization technologies;

- Extra CPU cycles are required to support I/O over virtualized devices which result in increased I/O access latency. This leads to reduction in the total usable bandwidth by the application hosted inside the virtual machine [9].
- Device virtualization architectures are cohesive because of which the device and its access software are monolithic. This causes performance degradation based on I/O workloads and also causes limitations on the scalability of VMs that can share the I/O device [2].

Both these reasons cause variability in application performance depending on the nature of workload and the number of virtual machines sharing the I/O device. One way to control this variability is to impose necessary Quality of Service (QoS) controls on resource allocation and usage of all resources, including the I/O devices, by the virtual machines. Ideally, the QoS controls should ensure that there is no loss of application performance when hosted on virtualized servers with shared resources.

The chapter begins with a discussion on the resource specific QoS controls that application performance depends on. An exploration of prevalent I/O virtualization architectures with existing QoS controls, with specific focus on the issues in sharing of a single NIC across multiple VMs, is presented to illustrate the previously described problems. This is followed by a proposal, which is an extension to the I/O device architecture, as recommended in the PCI-SIG IOV specification [3], for virtualizing network I/O devices. The aim of the proposal is to enable fine-grained controls to a virtual machine on the I/O path of a shared device so that there is no loss of application performance, at the same time minimizing loss of usable device bandwidth. The proposed architecture allows native access of I/O devices to virtual machines and provides device level QoS hooks for controlling VM specific device usage. For evaluating the architecture layered queuing network (LQN) [4][5] models are used. Simulations studies on the LQN models demonstrate the benefits of the proposed architecture. On the proposed architecture the application throughput observed is 60% more than what is observed on the existing architectures and the performance is closer to non-virtualized servers. The device-level QoS controls provide fine-grained controls on device usage by the sharing VMs and significantly improve the usable device bandwidth on the shared device. The proposed architecture also improves scalability in terms of the number of virtual machines intending to share the I/O device

2. Application requirements for performance isolation on shared resources:

Application performance is based on timely availability of the required resources like processors, memory and I/O devices. The basic guideline for consolidating enterprise servers over multi-core virtualized systems is by ensuring availability of required resources in a timely manner [6]. For the system to be able to do so, the application resource requirements can be enumerated in terms of resource tuples. A resource tuple is similar to a database record in the sense that it contains a sequence of values that describe specific attributes of a resource.

Table 1: An example resource tuple definition for a VM

```
</CPU_Resource_Descriptor_tuple>
        (Default_Speed=1800MHz:Min_speed=1500MHz:Max_speed=2000MHz)
        (Default_Count=4:Min_count=1:Max_count=4)
        (Default_L1Cache=64KB:Min_L1cache=64KB:Max_L1cache=64KB)
        (Default_L2Cache=512KB:Min_L2cache=64KB:Max_L2cache=512KB)
</end_of_tuple>
</Memory_Resource_Descriptor_tuple>
        (Default_Space=2000MB:Min_space=1000MB:Max_space=4000MB)
        (Default_Bandwidth=6400MBps:Min_bandwidth=6400MBps:Max_bandwidth=6400MBps)
</end_of_tuple>
</Network_Resource_Descriptor_tuple>
        (Default_Speed=1000Mbps:Min_speed=100Mbps:Max_speed=1000Mbps)
        (Default_Bandwidth=5000KBps:Min_Bandwidth=5000KBps:Max_Bandwidth=8000KBps)
</end_of_tuple>
</Disk_Resource_Descriptor_tuple>
        (Default_space=1000MB:Min_space=1000MB:Max_space=1000MB)
        (Default_bandwidth=100MBps:Min_bandwidth=100MBps:Max_bandwidth=400MBps)
</end_of_tuple>
```

In this context, a resource tuple lists the various resources that the virtual machine's performance depends on. Using the xml format for resource specification, akin to Globus Resource Specification Language [7], the following example in Table 1 illustrates the resource tuples for a typical VM that has both compute and I/O workloads. Each resource tuple is specified using a set of attributes which completely describe the specific resource requirement in terms of the quantity, number of units, speed of resource access and any other associated attributes of the resource. With reference to Table 1, the "CPU_Resource_Descriptor_tuple" describes the various attributes, like speed, count, L1 cache, L2 cache, of the CPU resource. Each attribute is described by three values, namely, Default, Minimum and Maximum. The Default parameter specifies the attribute value that the virtual machine monitor (VMM) can initially allocate to the VM. On an average, this is the value that the VM expects to use. The minimum parameter value defines the least value for the attribute that the VM could use while supporting the guaranteed performance and maximum attribute value defines the maximum amount of resource that the VM can use while supporting its workload. For example in the resource tuple depicted in Table 1, while the Default_speed of CPU expected is 1800MHz, the application can run with acceptable degraded performance using a CPU_speed of 1500MHz and with enhanced performance using a CPU_speed of 2000MHz. Ofcourse, all the three attributes can be effectively used if the VMM uses dynamic adaptive resource allocation policies.

In the context of multi-core servers with server consolidation as the goal, the resource tuples are assumed to be for each virtual machine of the virtualized server. This assumption is based on the fact that one application server is consolidated using one VM. In the case where multiple applications are co-located on a single VM, these resource tuples can be arrived at using the aggregated resource requirement of all the applications hosted by the VM. The resource tuples are used by the VMM while allocating or de-allocating resources to the VMs.

3. Prevalent Commodity Virtualization technologies and QoS controls for I/O device sharing: Virtualization technologies encompass a variety of mechanisms to decouple the system architecture and the user-perceived behavior of hardware and software resources. Among the prevalent technologies, there are two basic modes of virtualization, namely, full system virtualization like in *Vmware* [19] and para-virtualization as in *Xen* [14]. In full system virtualization complete hardware is replicated virtually. Instruction emulation is used to support multiple architectures. The advantage of this virtualization is that it enables unmodified operating systems (OS) to execute on the virtual machine (VM). Since it adopts instruction emulation, it tends to have high performance overheads as is observed in the experimental studies. Para-virtualization is more efficient and comes with lower performance overheads. In this kind of virtualization the OS is also modified suitably to run concurrently with other virtual machines on the same hardware. In either case, system virtualization is enabled by a layer called the virtual machine monitor (VMM) that now provides the resource management functionality across multiple virtual machines.

Commodity virtualization technologies like *Xen* and *Vmware* have made the normal desktop very versatile. A generic architecture of system virtualization implemented in such systems is given in Figure 1. The access to CPU resource is native, to all virtual machines sharing the CPU, for all instructions except the privileged instructions. The privileged instructions are virtualized, i.e., whenever such instructions are executed from within the VM, they are trapped and control is passed to the VMM. All I/O instructions fall under the privileged instructions and hence are trapped when executed from within the VM. Thus I/O devices like the Network Interface Card (NIC) and the DISK are treated differently when virtualized. There are two different, popularly adopted techniques, for I/O device virtualization. They are para-virtualization and emulation [17]. In para-virtualized mode of access, the hosting VM (as in *Xen*) or the VMM itself (as in *VMware-ESX* server) has exclusive access to the physical device. Other VMs sharing the device use software based mechanisms to access the physical device via the hosting VM or the VMM. In emulated mode of access, each VM sharing the physical device has a device driver that is implemented using emulation over the native device driver hosted by the VMM. Both these modes provide data protection and integrity to individual VMs but suffer from loss of performance and usable device bandwidth, as is illustrated by the evaluation in next section. To understand the effect of these different virtualization architectures on application performance, two different studies of *httperf* [13] benchmark are evaluated. The first is described in section 3.1 and explores how virtualization affects application performance. The second is described in section 3.2 which exposes the existing QoS constructs' effectiveness in providing the desired performance.

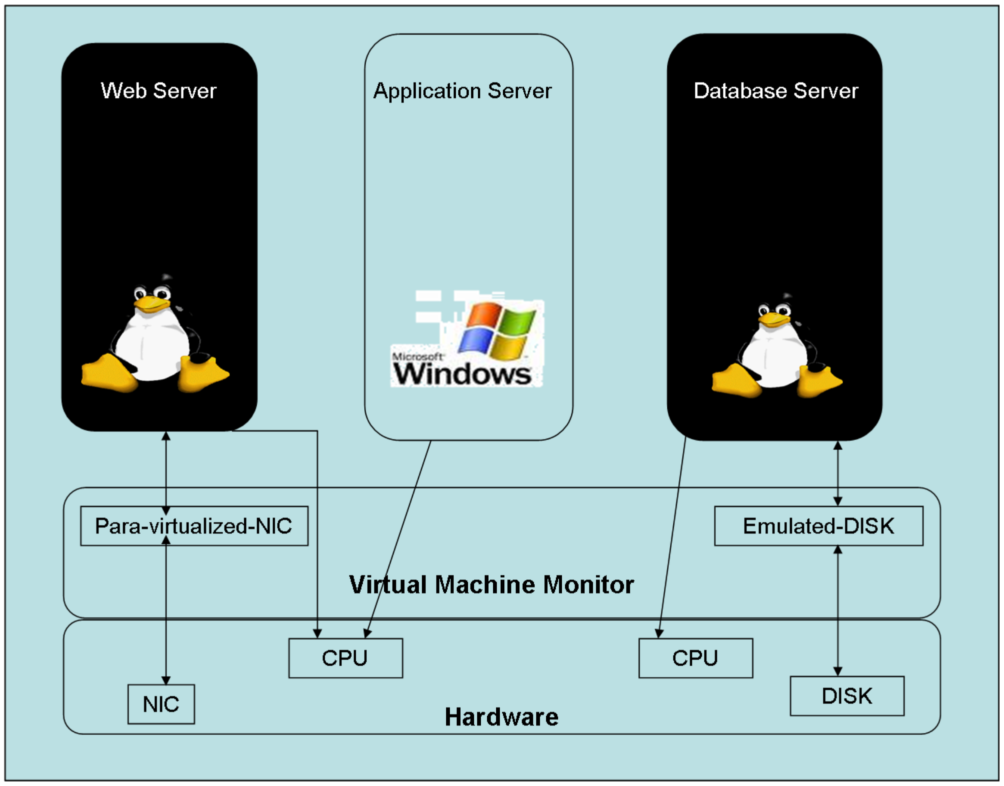


Figure 1: Generic System Virtualization architecture supported by prevalent commodity virtualization technologies.

3.1 Effect of Virtualization on Application Performance: The current commodity virtualization technologies like *Xen* and *VMware* allow for VM specific QoS controls on different resources using different mechanisms. The CPU resource allocations are handled directly by the VMM schedulers like *Credit*, *SEDF* or *BVT* schedulers of *Xen [14]*. Also, as discussed in [8] - [11] the virtualization architecture for these resources is fine-grained enough to give desired allocation and hence control the CPU allocation specific performance for CPU-based workloads. The problem is with I/O devices. The access to an I/O device is always through the OS kernel to ensure data integrity and protection. The device is never aware as to which VM is using it at any given instance of time; this information and control is managed by the VMM. Hence, resource allocation controls with regard to the I/O devices are at higher abstraction level rather than at the device level, unlike as in the case of CPU resource. This affects the way resource usage is controlled and thereby the application performance.

The way disk devices are shared is different from the way NICs are shared. In Unix like operating systems, to share disk devices, a disk partition is exposed as a file-system which is exported to a single VM. Any and every operation to this file-system is from a single VM and all read and write disk operations are block operations. The data movements to and from the file-systems are synchronized using the file-system buffer cache that is resident within the VM's address space. In para-virtualization the physical data movement is handled by the native device drivers within the VMM or hosted VM and the para-virtualized device driver of the VM. The overheads in this case are due to the multiple addresses translations of the data page between the device hosting VM and the application VM. For the emulation based device virtualization, overheads are due to the translation of every I/O instruction between the emulated and the native device driver. Due to this nature of I/O activity, VM specific file-system policies get to be implemented within the software layers of the VMM. Since the file-system activity is block based, setting up appropriate block sizes can, to some extent, enable the control of bandwidth and speed requirements on the I/O channel to the disk. However, these controls are still coarse-grained and can be insufficient for servers with high consolidation ratios.

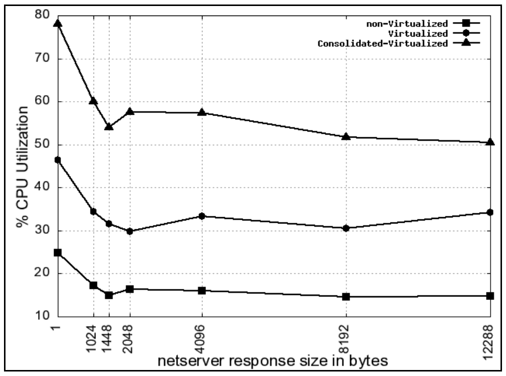
For network devices the existing architecture poses severe constraints. Network I/O is packet based and sharing a single NIC with multiple VMs has intermixed packet streams. This intermixing is transparent to the device and is sorted into VM specific streams by the VMM or the device hosting VM. Apart from this, every packet is subjected to either instruction translation or address translation due to virtualization. In either case the virtualization techniques build over existing "single-OS over single hardware" model.

To understand the effect of sharing a NIC at near maximum utilization bandwidth, on an application in virtualized environment, a comparative study of the behavior of two benchmarks, *netperf* [12] and *httperf* [13] for three different environments, namely non-virtualized, virtualized and consolidated-virtualized server is carried out. The non-virtualized environment is achieved by hosting the application on an independent server wherein all the hardware resources and software required by the application are available to it as unshared resources. This forms the baseline for the relation between the application resource requirement and achievable performance. The virtualized environment is achieved by transporting the non-virtualized server to a virtual machine hosted on a virtualized system. In this case, only one VM is hosted on the virtualized system. This environment exposes how virtualization effects an application, based on its workload. And, the consolidated-virtualized environment represents hosting of multiple VMs on a virtualized system with specific shared resources, in this case the NIC. In all the experiments described in this chapter, two VMs have been hosted under the consolidated-virtualized environment. The reason for choosing only two VMs and not more is because the focus of the experiments was to expose the issues in sharing virtualized I/O devices. Also, because of this, the scale of experiments, both in the real and simulation setup, was tractable for conclusive results especially because the benchmarks exercise the network interface.

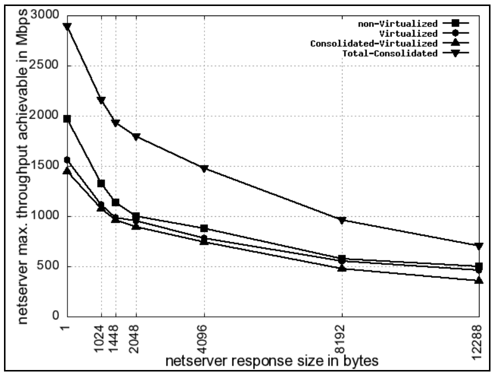
For the *netperf* benchmark, *netperf* is the name of the client and *netserver* is the server component. For this study, **TCP_CRR** test of *netperf* is chosen. The **TCP_CRR** test measures the connect-request-response sequence throughput achievable on a server and is similar to the access requests used in *http* based applications. In the case of *httperf* benchmark, the client, called *httperf*, communicates with a standard *http server* using the *http* protocol. The *httperf* client allows for specifying the workload in terms of the number of *http* requests to the server in one second, for a given period of time, to generate statistics like average number of replies received from the server, average response time of a reply and the network bandwidth utilized for the workload. While *netperf* gives the achievable or achieved throughput, *httperf* gives an average throughput calculated for a subset of samples, executed over a specified period of time, within the given experiment. Hence, *httperf* results give an optimistic estimate which may fall short of expectation in situations where sustained throughput is a requirement.

For this study, all the benchmarks were run on a *linux* server with a total of 2GB RAM and hosting a Broadcom Ethernet card configured to serve a bandwidth of 100Mbps. Although the NIC was capable of Gigabit speed, the network bandwidth was scaled down since this made the study of NIC sharing dynamics feasible since at higher request rates *httperf* reports failures that are not directly related to NIC sharing. The experimental values, plotted in the following charts, are an average of five independent measurements, to rule out the possibility of result variation. Each experiment was executed for a period of 600 seconds to represent steady state of server behavior. For the non-virtualized server, the machine is booted into a standard FedoraCore6 (FC6) Linux system. For the Xen virtualized server, the same system is booted into Xen3.0.3 with Fedora Core 6(FC6) for the Domain-0 and Domain-1. Domain-0 is the Independent Driver Domain(IDD) [15] hosting the network I/O device and Domain-1 is the virtual machine hosting the benchmark server. In case of the consolidated-virtualized server, two virtual machines, namely Domain-1 and Domain-2, are hosted on the machine along-with Domain-0. In this case, server component of the benchmark is hosted on each of the virtual machine and independent clients make requests to each of the servers. In the case of Vmware virtualized server, the virtualization is through VMware-ESXi hypervisor hosting Linux VMs.

For each of the benchmarks, the comparison is between a non-virtualized server with that of a virtualized server and consolidated-virtualized server. Two metrics are taken for comparison; the first one is %CPU utilization of the server and the second one reply rate or more commonly know as the application throughput of the server. Two charts, one for each of the metrics, are shown for each of the benchmarks. Figure 2 (a) shows the plot of %CPU utilization of the server against response message size for the *netperf* benchmark on a *Xen* server. In this chart we observe that moving from a non-virtualized to a virtualized server, the CPU resource utilization to support the same workload increases. This is the effect of virtualization on application resource requirement. Further, when we look at the case where two-VMs are consolidated on to a server, the server resource utilization increases significantly, as expected. In this case, each of the VMs sharing the same network interface, also share the *Xen*-VMM and IDD. When compared to servicing a single VM, the VMM and IDD now have to also service the requests of the second VM, which increases the CPU utilization on the server [9]. This is true for the *httperf* benchmark also, as indicated by the graphs in Figure 3(a). Also, the sharing of VMM and IDD by all device sharing VMs adds to device access latencies which contributes to a reduction in the maximum sustained throughput of the application. This is illustrated in Figure 2(b) and Figure 3(b), which give details of the throughput that *is* achieved in the case of *netperf* and *httperf* benchmarks.



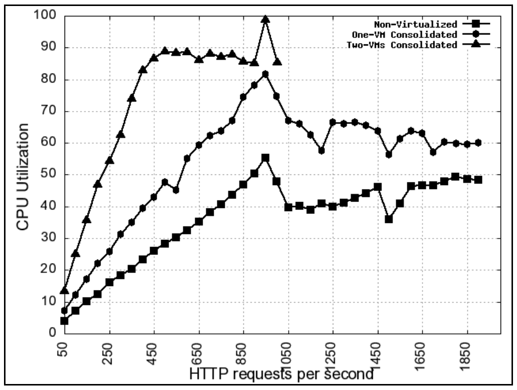
(a) *Server CPU Utilization*



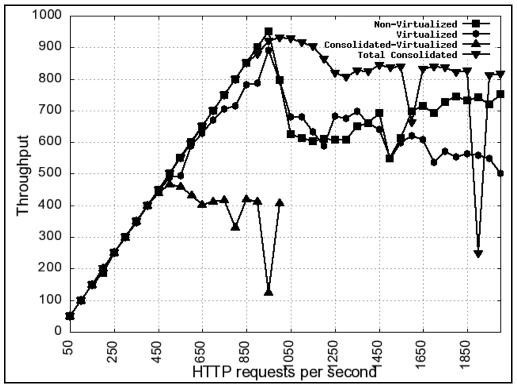
(b) *Server achievable throughput*

Figure 2: netperf server CPU Utilization and netserver achievable throughput on Xen-virtualized system

In the case of the *netperf* benchmark the loss of achievable throughput, for an application moving from non-virtualized to virtualized environment ranges from 4% to 20% and 10% to 29% in the case of consolidated-virtualized server. The surprising result is the case of total throughput achievable across both the VMs on a consolidated-virtualized server for *netperf*. We observe that the total throughput achievable is almost double that of what is achievable for a single VM. The reason for this is that the service time required to generate a response to the *netperf* request is long enough not to allow complete utilization of the network I/O bandwidth. And hence we see a better utilization for the consolidated server when two VMs are using the same NIC. This gives a clear motivation for sharing of I/O device in virtualized servers. However, in the case of *httperf* the improvement is not so dramatic. We observe here that while the reply rate increases linearly with the increase in request rate, till the peak rate, for the non-virtualized server, there is a gradual drop of throughput from 2% to 12.5% for the virtualized server as we progress from the request rate of 500reqs/s to the peak rate. The sustained total throughput for both the VMs, without loss, for the consolidated-virtualized server improves to 800reqs/s but still falls short of that achieved for the non-virtualized server by 10%.



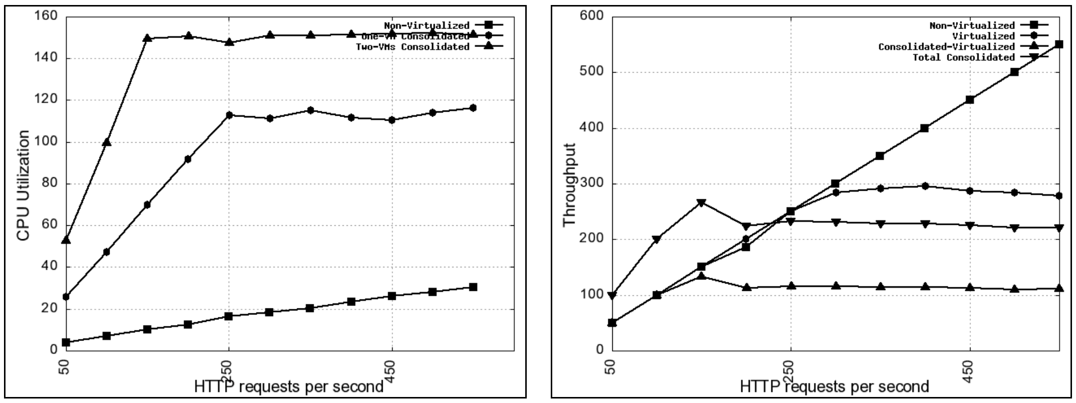
(a) *Server CPU Utilization*



(b) *Server throughput*

Figure 3: httperf characterization on Xen server

Similar behavior is observed when *Vmware* virtualization technology is used. This is depicted in Figure 4. The *httperf* benchmark tests using *VMware-ESXi* hypervisor were conducted on an *Intel Core2Duo* server with 2 cores. The speed of processor and NIC remain the same as that for the *Xen* server. Also, unlike in the case of *Xen*, pinning of *ESXi* server (the hypervisor) to a CPU is not possible. Hence, any CPU utilization measurements for the *ESXi* on *Vmware* show utilizations for all CPUs included, i.e. %CPU utilization can be above 100% in the case of multi-core systems. As for the VMs, both the VMs were hosted on the same physical CPU with one virtual CPU (VCPU) each. *VMware-ESXi* server implements NIC virtualization using device emulation. As can be seen, the overheads of emulation are comparatively high in relation to para-virtualization which is used in *Xen*. Here also, virtualization of NIC results in using up more CPU to support network traffic on a VM when compared to a non-virtualized server. The other important observation is the loss of application throughput. Device emulation imposes higher service times for packet processing and hence drastic drop of application throughput is observed, when compared to non-virtualized and para-virtualized systems. In this case we see almost 70% drop on maximum sustained throughput when compared to the throughput achieved in non-virtualized environment. This loss is visible even in the consolidated-virtualized case. Interestingly, the total network bandwidth used in the case of consolidated VMs was only 50% of the available bandwidth. Hence, the bottleneck is the CPU resource available to the VMs since each of the VM was hosted on the same core. When the same experiment was carried out with each VM pinned to an independent CPU, there was an increase in application throughput, but this throughput still fell short by 10% of what was achieved for the single VM. This indicates that even with the availability of required resources to each of the VMs the device sharing architecture in virtualized servers has constraints that impose severe restrictions in usable bandwidth and scalability of device sharing.



(a) Server CPU utilization (b) Server throughput

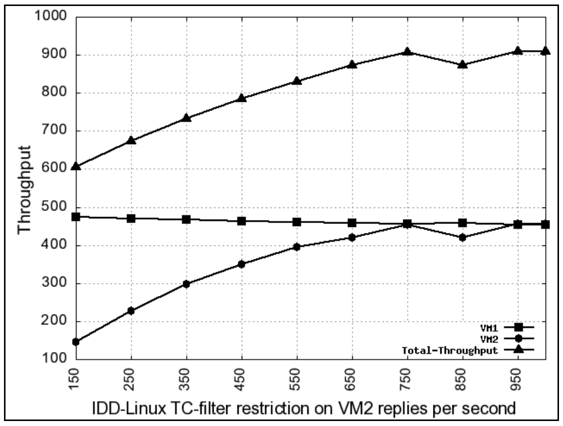
Figure 4: httperf characterization on Vmware-ESXi server

The most noteworthy point of observation is the behavior of each stream of benchmark in the case of consolidated-virtualized server. We see that in general there is a further reduction in throughput, when compared to the virtualized environment, in *netperf* and *httperf*, with a marked decrement in the later case. This indicates the obvious; lack of QoS constraints would lead to severe interference in performance offered by the VMs sharing a common device. It is noteworthy here that in *linux* packet level QoS controls are available as a software stack through the *netfilter* module, which is a serious drawback. While the software controls work well for the transmission path, the reception path is virtually uncontrolled at the device level. As a result, it may so happen that the device is receiving a large number of packets for a VM that would eventually be dropped and this leads to loss of bandwidth for the other VMs sharing this device. Provisioning for QoS controls in such systems is currently done only in software that is controlling the device, since the device is oblivious to the fact that it is being shared by multiple VMs. These controls are placed quite high in the software stack where the source or destination of the data is known. This causes the controls to be coarse-grained. In scenarios where I/O device utilization is pushed to its maximum, limitations of such QoS controls are revealed as loss of usable bandwidth or scalability of sharing. This results in unpredictable loss of application performance as is described in the next section.

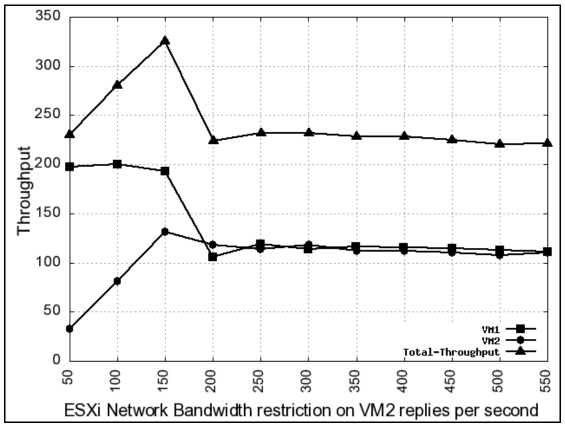
3.2 Evaluation of prevalent Network QoS controls available in Virtualized Servers: To understand the effect of software based QoS controls on network bandwidth sharing in virtualized servers, an analysis of experimental study conducted with network bandwidth control on a consolidated virtualized server, hosting two VMs, each running an application server and sharing a NIC. Each VM was built with with 1 VCPU, 512MB RAM, 25GB disk partition and a Virtual NIC built using a software bridge (*Xen*) or switch (*VMware-ESXi*) over a single physical NIC configured to 100Mbps speed. Each VM hosted an identical but independent http server. Initially the VCPUs of each VM were pinned to the same physical CPU. The experimental setup consisted of two independent httperf clients, each making requests to an independent http server. We choose to use *httperf* benchmark for this study and not *netperf* because the bandwidth control mechanisms available are based on time-sampled averages and hence need certain time specific observation to understand the mechanism control on the constrained channel. Initially it would seem that the control is not very effective but with the progress of time, once a steady state behavior sets in with respect to the traffic on the network channel, the effect of the control is visible on the application throughput.

To understand the NIC sharing dynamics, two distinct studies were made. The first was applying no bandwidth control on the NIC, each VM was subject to equal load from each of the httperf client. As can be observed from the performance of consolidated virtualized server case in Figure 3 and Figure 4 the NIC sharing is equal in both the virtualization solutions. When no QoS controls are enforced and each VM has equal demand for the resource, it is shared equally on a best effort basis. In the second study bandwidth control was enforced on one VM, namely VM2, while allowing complete available bandwidth to the other VM, the VM1. The bandwidth controls enforced were based on the following principle. For each of the virtualization technologies used, i.e. *Xen* and *Vmware,* the network bandwidth used by a single VM to support different httperf request rates, without performance loss, was measured. These bandwidth measurements were then used to apply control on the outgoing traffic from VM2 since the available controls allow constraints only on the outgoing traffic. On the incoming traffic, ideally the control should be applied at the device or the device driver. Such controls are not available currently. Instead, in Xen, atleast one can use the *netfilter* module stream based controls after receiving the packet. This is of not much use, because by receiving a packet that could potentially be dropped, the device bandwidth is anyway wasted. Hence, the study involves using only the outgoing traffic controls for the constrained VM. On the VM2 where the QoS control was applied, the study was made by systematically applying bandwidth control from 150 to 950 httperf requests/sec for the *Xen* server and from 50 to 550 httperf requests/sec for the *VMware* server. The selection of this different range for each of the virtualized server is based on the maximum throughput each can support in a consolidated virtualized environment. For each QoS control, the maximum throughput achieved, without loss, by each of the VM is plotted in the graphs of Figure 5. In these Figures, the x-axis represents the httperf request rate based on which the network bandwidth control was applied on the VM2. The y-axis represents the application throughput achieved by the VM. In the case of Xen, *linux tc* utility of the *netfilter* module was used to establish appropriate bandwidth controls. Specifically, each traffic stream from the VMs was defined using *htb class* with *tbf queue discipline* with the desired bandwidth control. Each *queue* was also setup with a burst value to support a maximum of 10 extra packets. In the case of *Vmware*, the *Veam Monitor* controls for network bandwidth were used and were populated with the same QoS controls as was done for the *Xen* server.

For both solutions, we notice that applying bandwidth control on one VM does not necessarily allow the availability of idle bandwidth to the unconstrained VM. Referring back to Figure 3 and Figure 4, we observe that the maximum application throughput achieved in the case of consolidated server for *Xen* is around 475 replies/s and for *VMware* is around 150 replies/s. Figure 5 demonstrates that even with QoS controls on one VM the other VM is not able to exploit the bandwidth availability. We also observe that the unconstrained VM does not improve the total throughput achieved across the shared interface. The reasons for this behavior are multitude, the most significant one being the virtualization overhead in terms of the CPU resource required by the VMM or the hosting VM to support I/O workload and serialization of this resource, second is the lack of control on the incoming network stream, third is that of the incoming stream having higher priority over the outgoing stream. All of these problems lead to unpredictable performance even with appropriate QoS controls in place.



(a) Xen-IDD Linux QoS controls on VM2



(b) Vmware-ESXi QoS controls on VM2

Figure 5: Effect of hypervisor network bandwidth controls on application throughput for consolidated virtualized server hosting two virtual machines.

As observed from the behavior of the benchmarks, we identify the following bottlenecks for sharing network I/O device across multiple VMs on *Xen* or *Vmware* virtualized server.

- Virtualization increases overheads in device utilization which leads to increased CPU utilization of the VMM and IDD hosting the device.
- Virtualization overheads cause loss of device bandwidth utilization from within a VM. Consolidation improves the device bandwidth utilization but further adds to CPU utilization of the VMM and IDD. Also, if the VMM and IDD do not support concurrent device access APIs they become the bottlenecks for the shared device access.
- QoS features for regulating incoming and outgoing traffic are currently implemented in the software stack. Uncontrolled incoming traffic to a VM sharing a network device can severely impact the performance of other VMs because the decision to drop an incoming packet is taken after the device has received the packet.

Based on the above observations, we propose an extension to I/O virtualization architecture, as recommended by the PCI-SIG IOV specification [3]. The PCI-SIG IOV specification defines the rudiments for making I/O devices virtualization aware. On the multi-core servers with server consolidation as the goal, particularly in the enterprise segment, being able to support multiple virtual I/O devices on a single physical device is a necessity. Already high speed network devices like 10Gbps NICs are appearing in the market. Pushing such devices to even 80% utilization needs fine-grained resource management at the device level. The basic goal of the proposed architecture is to be able to support finer levels of QoS guarantees, without compromising on device utilization. This proposal has its basis in exokernel's [16] implementation of I/O handling in the sense of decoupling device management mechanism from data access mechanism. The architecture is designed to enable native access of I/O devices to virtual machines and provide device level QoS hooks for controlling VM specific device usage. The architecture aims to reduce network I/O device access latency and enable improvement in effective usable bandwidth in virtualized systems by addressing the following issues:

- Separating device management issues from device access issues.
- Allowing native access of a device to a VM by supporting concurrent devices access and eliminating IDD from the path of device access.
- Enable fine-grained resource controls at the device.

The rest of the chapter is organized as follows. We describe the need for extending I/O device virtualization architecture by presenting the related work in section 4. In section 5 we highlight the bottlenecks of the device access path in the existing architecture, followed by a detailed description of the proposed architecture to overcome these bottlenecks taking *Xen* virtualization as the reference model. We then describe the network packet workflow for the proposed architecture, which forms the basis for generating the LQN model used in the simulation studies for architecture evaluation described in section 6. Brief description of the LQN model generation and detailed presentation of simulation results is covered in section 0. Finally, in section 8 we conclude with remarks on the benefits of the architecture.

4. Review of I/O virtualization techniques: I/O virtualization started with dedicated I/O devices assigned to a VM and has now evolved to device sharing across multiple VMs through virtualized software interfaces [17]. A dedicated software entity, called the I/O domain is used to perform physical device management. The I/O domain can be part of the VMM or be an independent domain, like the independent driver domain (IDD) of *Xen*. In the case of IDD the I/O devices are private to the domain and memory accesses by the devices are restricted to the IDD. Any application in a VM seeking access to the device has to route the request through the IDD and the request has to pass through the address translation barriers of the IDD and VM [22] - [27]. Recent publications on concurrent direct network access (CDNA) [20] and scalable self-virtualizing network interface are closer to the proposed work. However, the scalable self-virtualizing interface [21] 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 work is architecturally similar to the proposal in the paper. CDNA relies on per VM Rx/Tx ring buffers to manage VM specific network data. The VMM handles the virtual interrupts. However, it does not talk about the performance interference due to uncontrolled data reception by the device nor does it talk about 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 fine-grained control on interference effects. The proposed architecture has it basis in *exokernel's* 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, talks 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 [28], Neterion [29], NetXen [30], etc. CrossBow[31] 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 these have ignored the control of reception at the device level. We believe that lack of such a control on highly utilized devices will either cause performance degradation or lead to under-utilization of the device bandwidth.

5. Extension of I/O virtualization architecture: In the existing *Xen* virtualization architecture, when we analyze the network packet workflow we observe few bottlenecks which we aim to eliminate in the proposed scheme. The bottlenecks are:

- Since the device is shared, the device memory behaves like a common memory for all the contending VMs accessing the device. One misbehaving VM can ensure deprivation leading to data loss for another VM.
- The IDD is also a bottleneck for all the VMs sharing the device. IDD incurs processing overheads for each VM. Current IDD implementations do not have any hooks for controlling the overheads on a per VM basis. Lack of such controls leads to performance interference in the device sharing VMs.
- Every network packet has to cross the address translation barrier of VMM to IDD to VM and vice-versa. This happens because of lack of separation of device management issues from device access issues. The service overheads of this stage-wise data movement causes drop in effective utilized device bandwidth. In multi-core servers with scarce I/O devices, this would mean having high-bandwidth under-utilized devices or low throughput applications on the consolidated server.

To overcome the above listed drawbacks, we propose an extended architecture for virtualizing I/O devices that enables separation of device management issue from device access issue. This is done by building device protection mechanisms into the physical device and managed by the VMM. As an example, for the case of NIC, the VMM should be able to recognize the destination VM of an incoming packet by the interrupt raised by the device and forward it to the appropriate VM. The VM should then be able to process the packet as it would do so in the case of non-virtualized environment. Thus, 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.

5.1 Proposed I/O Virtualization Architecture Description: Figure 3 gives a block schematic of the proposed I/O virtualization architecture. The picture depicts a NIC card that can be housed within a multi-core server. The card has a controller that manages the DMA transfer to and from the device memory. The standard device memory is now replaced by a 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. 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 virtual-NICs that can be exported. Although, the finite number of physical resources on the NIC restricts the number of vNICs that can be exported, judicious use of native and para-virtualized access to the vNICs, based on the QoS guarantees a VM needs to honor, can overcome the 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. The VMM can aid in setting up the appropriate hosting connections based on the requested QoS requirements. The 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 construct to define a virtual device on a physical device as depicted in Figure 3. 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 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 the IDD layer for network I/O in Xen is replaced by a VM's native device driver. This device driver can only manipulate the restricted device address space which was exported through the virtual-NIC interface by the VMM.

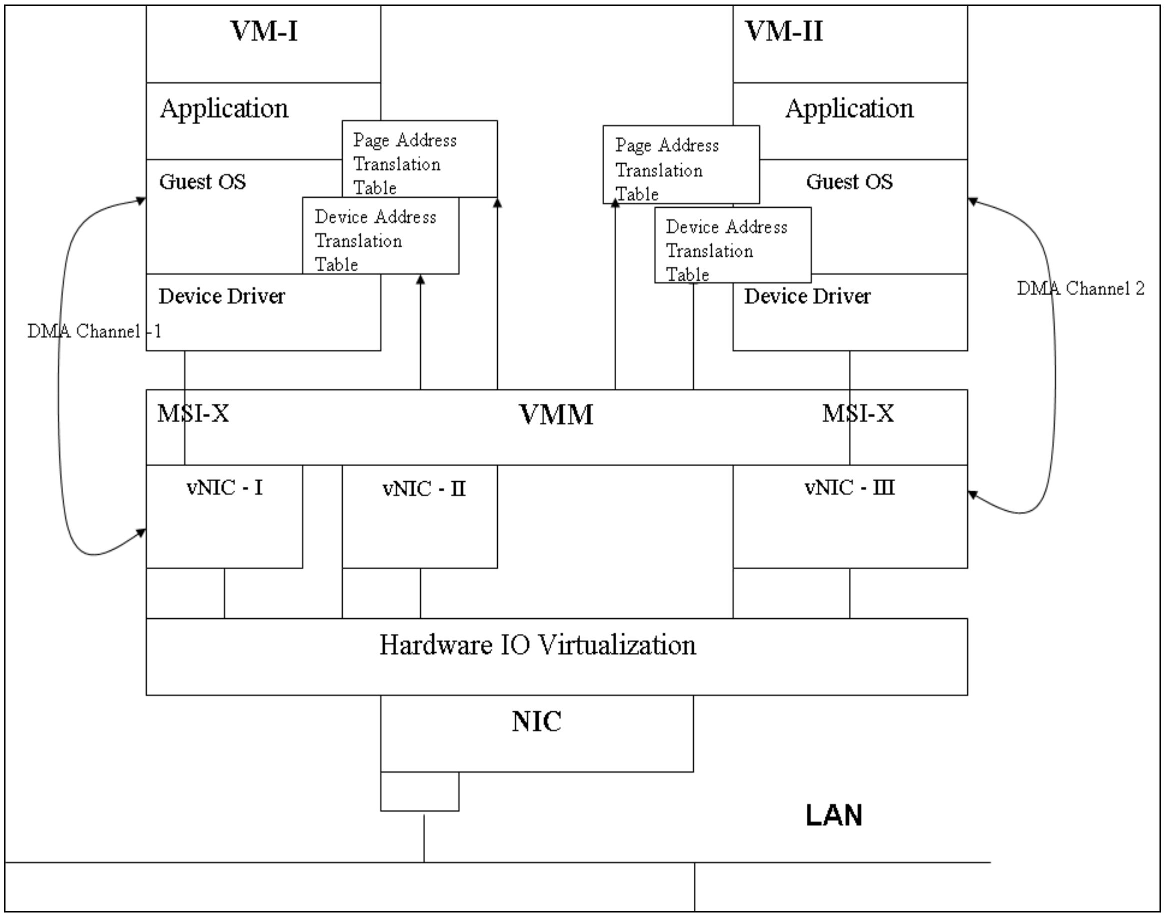
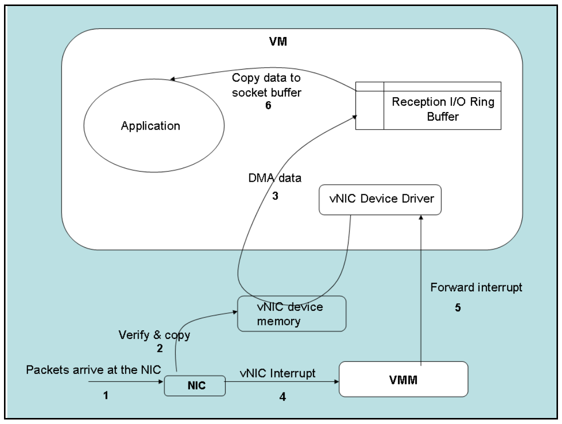

VM-I
VM-II
Application
Application
Page Address Translation Table
Page Address Translation Table
Guest OS
Guest OS
Device Address Translation Table
Device Address Translation Table
Device Driver
Device Driver
DMA Channel -1
DMA Channel 2
MSI-X
VMM
MSI-X
vNIC - I
vNIC - II
vNIC - III
Hardware IO Virtualization
NIC
LAN

Figure 6: NIC architecture supporting MSI interrupts with partitionable device memory, multiple device register sets and DMA channels enabling independent virtual-NICs.

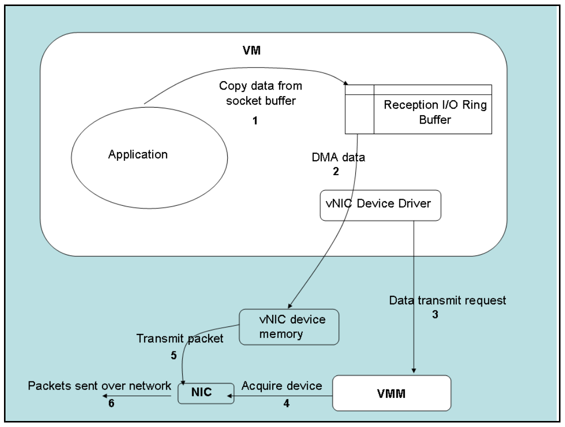
With the virtual-NIC, the VMM only identifies and forwards the device interrupt to the destination VM. The OS of the VM now handles the I/O access and thus can be accounted for the resource usage it incurs. This eliminates the performance interference due to IDD handling multiple VMs' request to a shared 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 virtual-NIC 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 does DMA 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 based processing 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 device memory free level. 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, as is done in the existing architectures.

5.2 Network Packet workflow using the virtualized I/O architecture: With the proposed I/O device virtualization architecture, each VM gets direct access to the shared I/O device without having to route the request through the IDD. Only the device interrupts get routed through the VMM. In Figure 4a, and Figure 4b the workflow for network data reception and transmission using the described device virtualization architecture is depicted. When a packet arrives at the NIC, it deciphers the destination address of the packet, checks if it is a valid destination, then copies the packet into the destination VM's portion of the device memory and issues DMA to the destination VM based on the virtual NIC's priority. On completion of the DMA the device raises an interrupt to the VMM. The VMM intercepts the interrupt, determines the destination VM, forwards the interrupt to the VM and schedules it. The VM's device driver then receives the data from the VM specific device descriptor rings as it would do in the case of non-virtualized server. In the case of transmission, the process is same as in the case of non-virtualized server, except that the VM's device driver DMA's data directly into the device memory allocated to its virtual-NIC. It may be worthwhile to note here that the code changes to support this architecture in the existing implementation will not be excessive. Each VM can use the native device driver for the exported virtual device interface. This device driver is the standard device driver for the IOV complaint devices with the only difference that it has now restricted device access. The device access restrictions in terms of memory, DMA channels, interrupt line and device register sets are setup by the VMM when the VM requests for a virtual device. With the virtual device interface the VMM now only has to implement the virtual device interrupts.



(a) Packet reception workflow



(b) Packet transmission workflow

Figure 7: Workflow of network I/O communication with improvised I/O device virtualization architecture.

6. Architecture Evaluation: Since the architecture involves design of a new NIC and a change in both VMM and VM device handling code, we first choose to evaluate the architecture using simulation based on layered queuing network (LQN) [18] models to understand the benefits. The reason for choosing LQN based modeling is twofold. One, there is a lack of appropriate system simulation tools that allow incorporating design of new hardware along-with VMM and VM OS changes. Second, LQN models are intuitive queuing models that enable capturing of the devices and software contention in the end-to-end workflow, right from the application to the device including the intermediate layers of the VM, IDD and VMM. With appropriate profiling tools, the LQN models are fairly easy to build and effective in terms of capturing the causes of bottlenecks in the access path. For further details on general description of LQN modeling and MOL refer [5].

6.1. LQN model for the proposed architecture: We generate the LQN model manually using the LQNDEF [4] software developed at the RADS lab of Carleton University. Here we present results for the model generated for the httperf benchmark since the bottleneck issues are prominent for this benchmark. Complete details on generating of the LQN models and validating the models against experimental data for this benchmark are discussed in [32] [33]. Three assumptions were made while generating the LQN models.

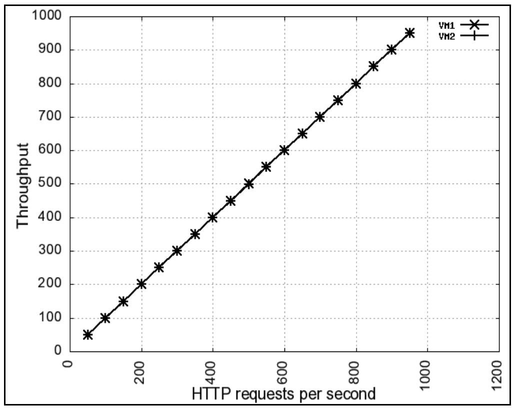
- The service times established at each of the entries constituting the LQN were populated based on the service times measured for an http request, instead of a tcp-packet. While it is feasible to model packet level contention, the reason for choosing request level contention was to be able to measure the benchmark throughput in terms of the number of satisfied requests. The model validation results demonstrate that there is no significant loss/gain (< 1%) of throughput because of this.
- For the chosen mode of run of httperf benchmark, the arrival request rate was observed to be uniform and measured service times were also uniform. Hence, the

service times and arrival rates populated on the LQN model were modeled as deterministic.

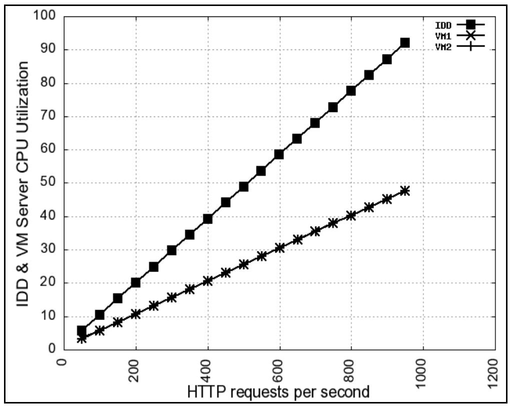
- For all the activities proposed to be handled by the device hardware in the proposed architecture and modeled as separate entities in the LQN model, the service time was set to be significantly low (1 X 10e-10 seconds). For the rest of the software entries, the service times were derived based on the measurements made for the non-virtualized servers. This is justified since our proposed architecture gives native access to the device from within the VM which is assumed to be running the same Guest OS as was used for the non-virtualized server.

In general it is observed that the maximum throughputs observed using the LQN model are higher than the experimental observations. The reason for this is simple. For every packet received or transmitted in Linux, there are several layers of the network stack that each packet has to pass through. The time taken to traverse this passage is what is recorded by the profiler as the service time. In real system, to match the difference between the device speed and CPU speed, appropriate memory buffers (TCP send and receive buffers of linux kernel) are maintained. The sizing of these buffers affects the observed application throughput. Normally, higher the buffer size larger the observed throughput. This trend is maintained to the point till the device is able to handle the network traffic. Once device saturation is reached, the failure behavior usually results in sudden drop in application throughput. While setting up the LQN model we choose to use the maximum permissible buffer size in the simulator (which is more than 3 times than what was set on the experimental system) so as to determine the maximum possible throughput for the setup service times. This gives an idea on the upper-bound of application throughput on a system with maximum possible resources with the service times possible with the architecture. The idea is to eliminate buffer size constraint in the simulation environment. While it is true that for the proposed architecture in which native access to the I/O device is provided, the maximum throughput that can be achieved in reality cannot exceed that of the maximum throughput achieved in the case of non-virtualized server, the results observed using simulations are contradictory. This is because of the fact that in the simulation environment, the buffers sizes used were maximum permitted. Hence, to make the comparison fair, care has been taken to generate results of existing architecture in a simulation environment, validate these results with those observed experimentally and then used for comparison with the results of the proposed architecture.

Simulation and Results: In order to evaluate the proposed architecture we use the *parasrvn* simulator of the LQNS software distribution [4] from Carleton University. Since we believe this architecture is more suitable and will have practical usefulness for multi-core servers, we evaluate such systems. For our study the LQN model consists of one VMM and two VMs and each of these is pinned to a different core. We validate the LQN model for the proposed architecture against the existing *Xen* architecture for a multi-core server. To observe the behavior on a multi-core server for the existing architecture, in the LQN model the VMM/IDD and each of the VMs are placed on an independent core. Figure 5 depicts the results of achievable throughput and server CPU utilization for a multi-core server with two VMs consolidated. The throughput graph for both the VMs is similar and appears overlapped in the chart. As can be noted from *Figure 8*, in a multi-core environment with *Xen* IDD, VM1 and VM2 each pinned to a core, and each VM servicing one httperf stream, the maximum throughput, without loss, achievable per stream is 950requests/s as against 450requests/s in the case of single-core. But, for the maximum throughput, we observe that the Xen-IDD, which is hosting the NIC of the server, the CPU utilization saturates. This indicates that further increase in request rate is not possible since the processor core serving the Xen-IDD does not have any computing power left. *Figure 9* shows these statistics for a similar situation but with the proposed architecture. We observe that the maximum throughput achievable now per stream increases to 1500 req/s, which is an increase of about 60% more. The total throughput achievable at the NIC, derived from consolidating the throughput of both the streams, also increases by 60% when compared to what was achieved on the existing architecture.

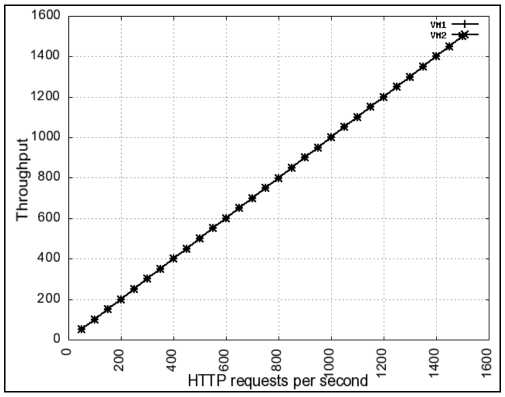


(a) Throughput

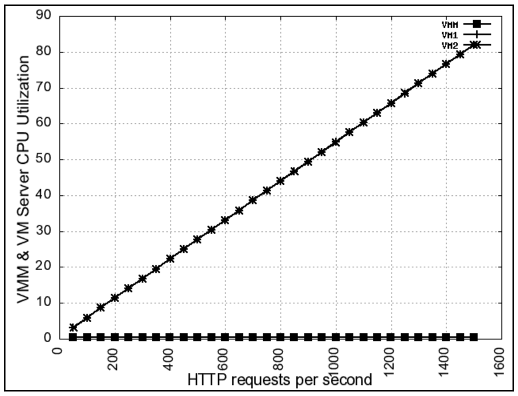


(b) CPU Utilization

Figure 8: Charts for maximum throughput achievable per httperf stream and CPU utilization for existing Xen architecture on a multi-core server hosting two VMs each servicing one of the httperf stream. The IDD, VM1 and VM2 are pinned to independent cores.



(a) Throughput



(b) CPU Utilization

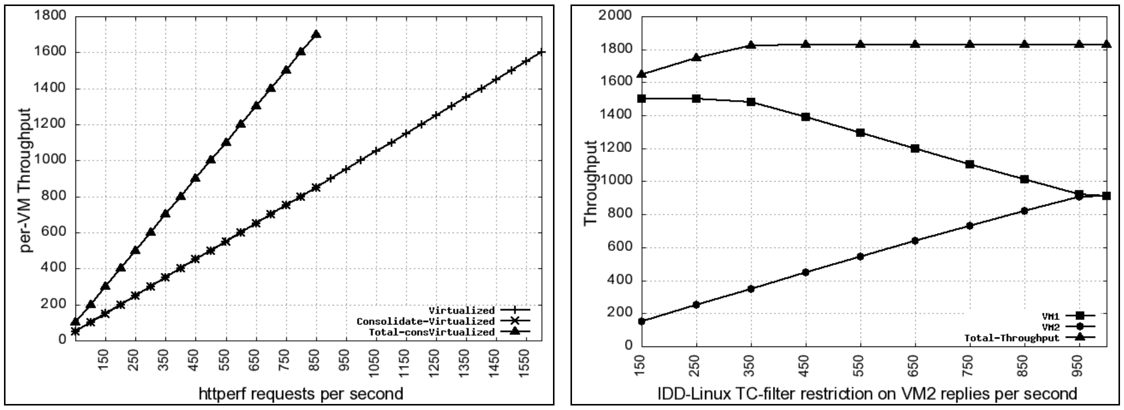
Figure 9: Maximum achievable throughput and CPU utilization charts for a multi-core Xen server incorporating the proposed I/O virtualization architecture and hosting two VMs, pinned to different cores, each servicing one httperf stream.

If we look at the CPU utilization of each VMs, we observe that the Dom0 which is the VMM for the NIC, now consumes very less CPU. The reason for this is that, the NIC is now offloading the identity of the destination of the packet and this identification happens at hardware speeds. To account for this, a very low value of service time is assigned in the LQN model. Also, in the existing model, bridging software that routes the packets to a VM and which has substantial overhead, is done away with in the proposed architecture. The net result is improved throughput, reduced virtualization overhead, and reduction of VMM/IDD resource consumption on behalf of VMs. We also notice that the VMM is now spending almost constant time which results in eliminating the performance interference. This also improves the scalability of sharing the device across VMs. With this architecture each VM is now accountable for all the resource consumption, thereby leading to better QoS controls.

We next evaluate the proposed architecture for QoS controls on the network bandwidth. Since this architecture is implemented using LQNs model, we make certain assumptions to simulate the network bandwidth controls as implemented in the *netfilter* module of *Linux*. LQNs is basically a queuing model wherein at any node (also called entry in *parasrvn* notation) of the queue is described using three parameters, namely, the arrival rate, the service time and the think time. The arrival rate models the rate of input requests at the entry, service time depicts the time the entry takes to process the request before forwarding to the next entry or replying back to the requesting entry and think time denotes the time before which the entry actually services the request. The think time parameter is useful to model policies like bandwidth restrictions, time-sharing intervals, periodic processing, etc. The LQN is basically a directed acyclic graph that captures the complete workflow. Hence, the arrival rate is set for the source entry and in this case represents the rate of request arrival at the network interface of the virtualized server. The service time represents the CPU time used for servicing the request by the entry of LQN and think time is used as the bandwidth restriction that was applied in experimental case. For example, to model 250reqests/second bandwidth restriction the think time derived is 1/250 seconds. This ensures that the entry will only process 250 requests/second and anything extra will be queued or dropped. To model this restriction within the experimental setup we use the burst parameter of the bandwidth control mechanism in Linux *netfilter* module.

The justification is that in the Linux *netfilter* module the bandwidth restriction manifests as packet loss after the restriction is saturated. The bandwidth restriction also has a burst parameter that allows for some extra packets delivery on the channel above the restriction. By setting the burst rate sufficiently low, in our case equivalent to 10 packets which is also the minimum that is permissible, we ensure that overflow of bandwidth on the constrained channel is restricted. For the html page that is requested in our experiments, total number of packets needed to complete a successful request is about 14. Hence by setting the burst rate to 10 packets ensures that the request fails thereby throughput reported takes into account the desired behavior. The think time setting in LQN model is more restrictive than the *netfilter*. But, since the think time value is based on the deterministic request rate parameter to define the bandwidth constraint, we still have equivalent results and this has been validated against observed values. The following graphs in *Figure 10* depict the effect of not imposing (*Figure 10* (a)) and imposing network bandwidth QoS on the incoming stream of VM2 (*Figure 10* (b)), in the proposed architecture. These simulations were conducted on a single core server to keep the throughput range such that time consumed for comparable results was minimized.

As can be observed from the graphs of *Figure 10* (a), for the best effort service, i.e. when no QoS constraints are applied on any of the sharing streams, the maximum throughput, without loss, achieved on any of the VMs is 850 replies/second on the consolidated-virtualized server. This is slightly less than half of what is achieved by the VM using an unshared device on the virtualized server (refer Figure *Figure 9*(a)). The graphs of *Figure 10* (b) show that, unlike as in the case of existing architectures, the QoS constraints when moved to device level, allow the usage of available bandwidth by the unconstrained channel. In this figure, VM2 is constrained to allow requests starting from 150 requests/second to 950 requests/second and VM1 is unconstrained. Since the NIC is discarding requests to VM2 that are above the specified request rate, VM1 is able to use the available bandwidth, hence we see higher throughputs (1500 replies/sec) on VM1. As the bandwidth control on VM2 is relaxed we see that the two graphs start converging towards each other and finally merge to the best effort case. The bandwidth control on the incoming stream also works to advantage on the http traffic because by discarding the request at the device itself, the server and hence the resources, are spared to respond on requests that will eventually be dropped because of bandwidth controls. The other observation is that when multiple VMs are sharing the NIC, the maximum bandwidth achievable on the unconstrained channel is less (<10%) than that which is achieved by the isolated VM. Further reduction on this loss is possible by applying channel based prioritization and bandwidth control on the outgoing channel of the constrained VM. The outgoing channel constraints are easily achievable by using existing mechanisms.



(a) Best effort, no Qos on NIC sharing. (b) QoS controls on VM2 outgoing channel.

Figure 10: Throughput achieved before and after imposing QoS controls on VM2 of the proposed architecture.

8. Conclusion: In this chapter 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. The architecture evaluation was done by capturing it as an LQN model and analyzing using simulation of the model. The simulation results show a utilization benefit of about 60%, without enforcing any QoS guarantees or applying any of the software optimization techniques to the I/O path. The proposed architecture also improves the scalability of VMs sharing the NIC. We also demonstrated 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. Although the evaluation was done for para-virtualized systems like Xen, we believe the ideas proposed would benefit fully virtualized systems like Vmware too. The reason being, in any of these virtualization techniques I/O device sharing is currently regulated through a common software entity, which is eliminated in the proposed architecture.

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] Robert P Goldberg, "Survey of Virtual Machine Research", IEEE-Computer, 1974, vol7, no.6, p34-45.

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

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

[5] J.A. Rolia and K. C. Sevcik, "The Method of Layers", IEEE Transactions on Software Engineering, Vol. 21, No.8, Aug 1995.

[6] Ben Verghese, Anoop Gupta and Mendel Rosenblum, Performance Isolation: Sharing and Isolation in Shared-Memory Multiprocessors, ACM SIGPLAN Nov 19, 1998.

[7] "The Globus Resource Specification Language RSL v1.0." [Online]. Available: http://www-fp.globus.org/gram/rsl spec1.html

[8] 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.

[9] 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.

[10] 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).

[11] 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).

[12] Rick A. Jones, 'netperf: A Network Performance Benchmark,' Revision 1. Information Networks Division, Hewlett-Packard Co., March 1993.

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

[14] 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.

[15] 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.

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

[17] 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.

[18] C. M. Woodside, J. E. Neilson, D. C. Petriu and S. Majumdar, "The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software", IEEE Trans. on Computers, vol. 44, no. 1, January 1995, pp. 20-34.

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

[20] 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).

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

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

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

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

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

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

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

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

[33] 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.