

Chapter 2

Quality of Service for I/O Workloads in Multicore Virtualized Servers

J. Lakshmi and S.K. Nandy

Abstract Emerging trend of multicore servers promises to be the panacea for all data-center issues with system virtualization as the enabling technology. System virtualization allows one to create virtual replicas of the physical system, over which independent virtual machines can be created, complete with their own, individual operating systems, software, and applications. This provides total system isolation of the virtual machines. Apart from this, the key driver for virtualization adoption in data-centers will be safe virtual machine performance isolation that can be achieved over a consolidated server with shared resources. This chapter identifies the basic requirements for performance isolation of virtual machines on such servers. The consolidation focus is on enterprise workloads that are a mix of compute and I/O intensive workloads. An analysis of prevalent, popular system virtualization technologies is presented with a view toward application performance isolation. Based on the observed lacunae, an end-to-end system virtualization architecture is proposed and evaluated.

2.1 Introduction

System virtualization on the emerging multicore servers is a promising technology that has solutions for many of the key data-center issues. Today's data-centers have concerns of curtailing space and power footprint of the computing infrastructure, which the multicore servers favorably address. A typical multicore server has sufficient computing capacity for aggregating several server applications on a single physical machine. The most significant issue with co-hosting multiple-server applications on a single machine is with the software environment of each of the appli-

J. Lakshmi (✉) · S.K. Nandy
SERC, Indian Institute of Science, Bangalore 560012, India
e-mail: jlakshmi@serc.iisc.ernet.in

S.K. Nandy
e-mail: nandy@serc.iisc.ernet.in

M. Cafaro, G. Aloisio (eds.), *Grids, Clouds and Virtualization*,
Computer Communications and Networks,
DOI [10.1007/978-0-85729-049-6_2](https://doi.org/10.1007/978-0-85729-049-6_2), © Springer-Verlag London Limited 2011

cations. System virtualization addresses this problem since it enables the creation of 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 results in complete software isolation of the VMs, which allows independent applications to be hosted within independent virtual machines on a single physical server.

Apart from the software isolation, the key driver for virtualization adoption in data-centers will be safe 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 that have both CPU and I/O workload components. Current commodity multicore technologies have system virtualization architectures that provide CPU workload isolation. The number of CPU-cores in comparison to I/O interfaces is high in multicore servers. This results in the sharing of I/O devices among independent virtual machines. As a result, this changes the I/O device sharing dynamics when in comparison to dedicated servers, wherein all the resources like the processors, memory, I/O interfaces for disk and network access are architected to be managed by a single OS. On such systems, solutions that optimize or maximize the application usage of the system resources are sufficient to address the performance of the application. When multiple, independent applications are consolidated onto a multicore server, using virtual machines, performance interference caused due to shared resources across multiple VMs adds to the performance challenges. The challenge is in ensuring performance of the independent I/O intensive applications, hosted inside isolated VMs, on the consolidated server while sharing a single I/O device [10].

Prevalent virtualization architectures suffer from the following distinct problems with regard to I/O device virtualization;

1. Device virtualization overheads are high due to which there is a reduction in the total usable bandwidth by an application hosted inside the VM.
2. Prevalent device virtualization architectures are such that sharing of the device causes its access path also to be shared. This causes performance degradation that is dependent on I/O workloads and limits scalability of VMs that can share the I/O device [1].
3. Device access path sharing causes security vulnerabilities for all the VMs sharing the device [35].

These reasons cause variability in application performance that is dependent on the nature of consolidated workloads and the number of VMs 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 shared resources. Ideally, the QoS controls should ensure that:

- There is no loss of application performance when hosted on virtualized servers with shared resources.
- Any spare resource is made available to other contending workloads.

The chapter starts with a discussion on the resource specific QoS controls that an application's performance depends on. It then explores the QoS controls for re-

source allocation and usage in prevalent system virtualization architectures. The focus of this exploration is on the issues of sharing a single NIC across multiple virtual machines (VMs). Based on the observed lacunae, an end-to-end architecture for virtualizing network I/O devices is presented. The proposed architecture is an extension to that of what is recommended in the PCI-SIG IOV specification [21]. The goal of this architecture is to enable fine-grained controls to a VM on the I/O path of a shared device leading to minimization of the loss of usable device bandwidth without loss of application performance. The proposed architecture is designed to allow native access of I/O devices to VMs and provides the device-level QoS controls for managing VM specific device usage. The architecture evaluation is carried out through simulation on a layered queuing network(LQN) [3, 4] model to demonstrate its benefits. The proposed architecture improves application throughput by 60% as in comparison to what is observed on the existing architectures. This performance is closer to the performance observed on nonvirtualized servers. The proposed I/O virtualization architecture meets its design goals and also improves the number of VMs that can share the I/O device. Also, the proposed architecture eliminates some of the shared device associated security vulnerabilities [35].

2.2 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 multicore virtualized systems is by ensuring availability of required resources as and when required [7]. For the system to be able to do so, the application resource requirements are enumerated using resource requirement (RR) tuples. An RR tuple is an aggregated list of various resources that the application's performance depends on. Thus RR tuple is built using individual resource tuples. Each resource tuple is made up of a list of resource attributes or the attribute tuples. Using this definition, a generic RR tuple can be written as follows:

$$\begin{aligned} \text{Application}(RR) = & \\ & (R1 < A1(\text{Unit}, \text{Def}, \text{Min}, \text{Max}), A2(\text{Unit}, \text{Def}, \text{Min}, \text{Max}), \dots >, \\ & R1 < A1(\text{Unit}, \text{Def}, \text{Min}, \text{Max}), A2(\text{Unit}, \text{Def}, \text{Min}, \text{Max}), \dots >, \\ & \dots) \end{aligned}$$

where:

- *Application(RR)*—Resource requirement tuple of the application.
- *R1*—Name of a resource, viz. processor (CPU), memory, network(NIC), etc.
- *A1*—Name of the attribute of the associated resource. As an example, if *A1* represents the CPU speed attribute, it is denoted by the tuple that describes the CPU speed requirements for the application.

- (*Unit, Def, Min, Max*) represent the *Unit* of measurement, *Default*, *Minimum*, and *Maximum* values of the resource attribute.

Using the XML format for resource specification, akin to Globus Resource Specification Language [13], the following example in Fig. 2.1 illustrates the application(RR) for a typical VM that has both compute and I/O workloads.

In the depicted example, the resource tuple for the CPU resource is described by the `<CPU_Resource_Descriptor>` and `</CPU_Resource_Descriptor>` tag pair. Attribute tuples relevant to and associated with this resource are specified using the attribute and value tag pair, within the context of resource tag pair. Each attribute is specified by its *Unit of measurement*, *Default*, *Minimum*, and *Maximum* values that the virtual machine monitor's (VMM's) resource allocator uses for allocating the resource to the VM. In the example, the CPU speed is defined by the attribute tags `<Speed>` and `</Speed>`. The *Unit of Measurement* for CPU speed is mentioned as MHz. The attribute values for *Default*, *Minimum*, and *Maximum* specify the CPU speed required for the desired application performance hosted inside the

<code><Application_RR_descriptor></code>	<code><Network_Resource_Descriptor></code>
<code><CPU_Resource_Descriptor></code>	<code><Speed></code>
<code><Speed></code>	<code><Unit>Mbps</Unit></code>
<code><Unit>MHz</Unit></code>	<code><Default>1000</Default></code>
<code><Default>1800</Default></code>	<code><Minimum>100</Minimum></code>
<code><Minimum>1500</Minimum></code>	<code><Maximum>1000</Maximum></code>
<code><Maximum>2000</Maximum></code>	<code></Speed></code>
<code></Speed></code>	<code><Bandwidth></code>
<code><NCPU></code>	<code><Unit>KBps</Unit></code>
<code><Default>4</Default></code>	<code><Default>5000</Default></code>
<code><Minimum>1</Minimum></code>	<code><Minimum>5000</Minimum></code>
<code><Maximum>4</Maximum></code>	<code><Maximum>8000</Maximum></code>
<code></NCPU></code>	<code></Bandwidth></code>
<code><L1Cache></code>	<code><Unit>KB</Unit></code>
<code><Unit>KB</Unit></code>	<code><Default>64</Default></code>
<code><Default>64</Default></code>	<code><Minimum>64</Minimum></code>
<code><Minimum>64</Minimum></code>	<code><Maximum>64</Maximum></code>
<code><Maximum>64</Maximum></code>	<code></Bandwidth></code>
<code></L1Cache></code>	<code></Network_Resource_Descriptor></code>
<code></CPU_Resource_Descriptor></code>	<code><Disk_Resource_Descriptor></code>
<code><Memory_Resource_Descriptor></code>	<code><Size></code>
<code><Size></code>	<code><Unit>MB</Unit></code>
<code><Unit>MB</Unit></code>	<code><Default>1000</Default></code>
<code><Default>2000</Default></code>	<code><Minimum>1000</Minimum></code>
<code><Minimum>1000</Minimum></code>	<code><Maximum>1000</Maximum></code>
<code><Maximum>4000</Maximum></code>	<code></Size></code>
<code></Size></code>	<code><Bandwidth></code>
<code><Bandwidth></code>	<code><Unit>MBps</Unit></code>
<code><Unit>MBps</Unit></code>	<code><Default>100</Default></code>
<code><Default>6400</Default></code>	<code><Minimum>100</Minimum></code>
<code><Minimum>6400</Minimum></code>	<code><Maximum>400</Maximum></code>
<code><Maximum>6400</Maximum></code>	<code></Bandwidth></code>
<code></Bandwidth></code>	<code></Disk_Resource_Descriptor></code>
<code></Memory_Resource_Descriptor></code>	<code></Application_RR_Descriptor></code>

Fig. 2.1 An example Application Resource Requirement tuple for a VM, expressed in XML

VM. *Default* value specifies the attribute value that the VMM can initially allocate to the VM. On an average, this is the value that the VM is expected to use. The *Minimum* value defines the least value for the attribute that the VM needs to support the guaranteed application performance. The *Maximum* value defines the maximum attribute value that the VM can use while supporting its workload. All the three attribute values can be effectively used if the VMM uses dynamic adaptive resource allocation policies. For each resource, its tuple is specified using attribute value tuples that completely describe the specific resource requirement in terms of the quantity, number of units, and speed of resource access.

On a virtualized server the physical resources of the system are under the control of the VMM. The resource tuples are used by the VMM while allocating or deallocating resources to the VMs. It can be assumed that RR contains values that are derived from the application’s performance requirements. In the context of multicore servers, with server consolidation as the goal, each application can be assumed to be hosted in an independent VM which encapsulates the application’s environment. Hence, the application’s resource tuples can be assumed to be the RR for each VM of the virtualized server. In the case where multiple applications are co-hosted on a single VM, these resource tuples can be arrived at by aggregating the resource requirements of all the applications hosted by the VM.

2.3 Prevalent Commodity Virtualization Technologies and QoS Controls for I/O Device Sharing

Commodity virtualization technologies like *Xen* and *Vmware* have made the normal desktop very versatile. A generic architecture of system virtualization, implemented in these systems, is given in Fig. 2.2. The access to CPU resource is native, to all VMs sharing the CPU, for all instructions except the privileged instructions. The privileged instructions are virtualized, i.e., whenever such instructions are executed

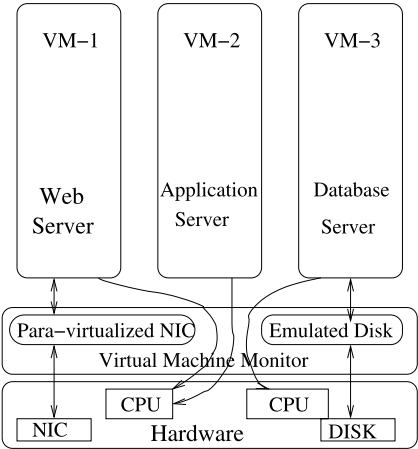


Fig. 2.2 Generic System Virtualization architecture of prevalent commodity virtualization technologies

from within the VM, they are trapped, and control is passed to the VMM. All I/O instructions fall under the category of privileged instructions. Thus, I/O devices like the Network Interface Card (NIC) and the DISK are treated differently, when virtualized. There are two different, popularly adopted, methods used for virtualizing I/O devices, namely, para-virtualization and emulation [27]. Para-virtualized mode of access is achieved using a virtual device driver along with the physical device driver. A hosting VM or the VMM itself has exclusive, native access to the physical device. Other VMs sharing the device use software-based mechanisms, like the virtual device driver, 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 independent VMs but suffer from loss of performance and usable device bandwidth. Details of the evaluation are elucidated in the following section. In order to understand the effect of the device virtualization architectures on application performance, experimental results of well-known benchmarks, *httperf* [8] and *netperf* [2], are evaluated. The first experiment is described in Sect. 2.3.1 and explores how virtualization affects application performance. The second experiment, described in Sect. 2.3.2, evaluates the existing QoS constructs in virtualized architectures for their effectiveness in providing application-specific guarantees.

2.3.1 Effect of Virtualization on Application Performance

Prevalent commodity virtualization technologies, like *Xen* and *Vmware*, are built over system architectures designed for single OS access. The I/O device architectures of such systems do not support concurrent access to multiple VMs. As a result, the prevailing virtualization architectures support I/O device sharing across multiple VMs using software mechanisms. The result is device sharing along with its access path. Hence, serialization occurs at the device and within the software layers used to access the device.

In virtualized servers, disk devices are shared differently compared to sharing of NICs. In the case of disk devices, a disk partition is exposed as a filesystem that is exported to a single VM. Any and every operation to this filesystem is from a single VM, and all read and write disk operations are block operations. The data movements to and from the filesystem is synchronized using the filesystem buffer cache that is resident within the VM's address space. The physical data movement is coordinated by the native device drivers within the VMM or the hosted VM, and the para-virtualized or emulated device driver resident in the VM. In the para-virtualized mode, the overheads are due to the movement of data between the device hosting VM and the application VM. In the case of emulation mode of access, the overheads manifest due to the translation of every I/O instruction between the emulated device driver and the native device driver. Due to this nature of I/O activity, VM specific filesystem policies get to be implemented within the software layers of

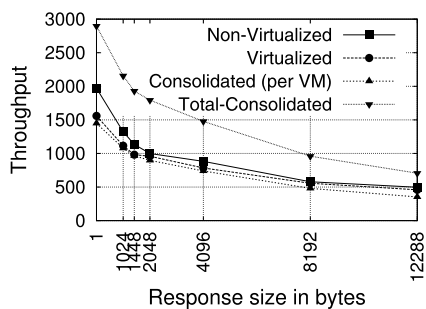
the VMM or the hosting VM. Since the filesystem activity is block based, setting up appropriate block sizes can, to some extent, enable the control of bandwidth and speed requirements on the I/O channel to the disk. However, these controls are still coarse-grained and are insufficient for servers with high consolidation ratios.

For network devices, the existing architecture poses different constraints. Unlike for the disk I/O which is block based, 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 per VM stream by the VMM or the hosting VM. Apart from this, every packet is subjected to either instruction translation (emulation) or address translation (para-virtualization) due to virtualization. In both the cases, virtualization techniques build over existing “single-OS over single hardware” model. This degrades application performance.

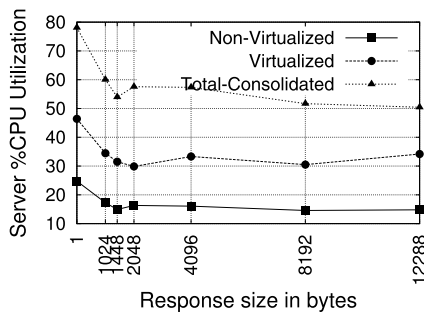
Throughput studies of standard enterprise benchmarks highlight the effects of virtualization and consolidation based device sharing. Since NIC virtualization puts forth the basic issues with virtualization technologies, an analysis of NIC sharing over application throughput is presented. Figures 2.3a and 2.4a depict the performance of two standard benchmarks, *netperf* [2] and *httperf* [8], wherein the benchmark server is hosted in three different environments, namely nonvirtualized, virtualized, and consolidated servers. The nonvirtualized environment is used to generate the baseline for the metric against which the comparison is made for the performance on virtualized and consolidated server. The virtualized server hosts only one VM wherein the complete environment of the nonvirtualized server is reproduced inside the VM. This environment is used to understand the overheads of virtualization technology. The consolidated server hosts two VMs, similar to the VM of the virtualized server, but with both VMs sharing the same NIC. The consolidated server environment is used to understand the I/O device sharing dynamics on a virtualized server.

For the *netperf* benchmark, *netperf* is the name of the client, and *netserver* is the server component. The study involves execution of the TCP_CRR test of *netperf*. The TCP_CRR test measures the connect-request-response sequence throughput achievable on a server and is similar to the access request 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. In the *httperf* test used, the 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 the average number of replies received from the server (application throughput), the 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.

It is observed from the throughput graphs of *netperf* and *httperf* that there is a significant drop in application throughput as it is moved from nonvirtualized to *Xen* virtualized server. *Xen* virtualization uses para-virtualization mechanism with

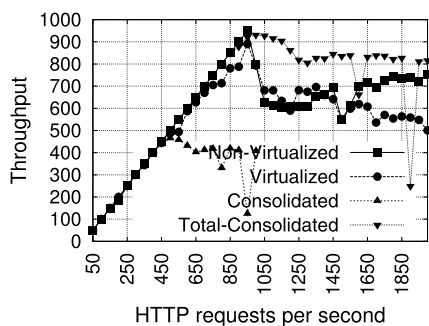


(a) Server achievable Throughput.

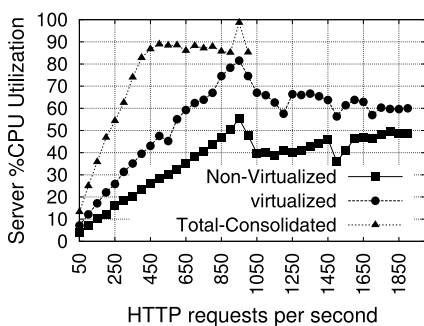


(b) Server CPU Utilization.

Fig. 2.3 *netserver* achievable Throughput and corresponding %CPU Utilization on *Xen* virtualized platform



(a) Server Throughput



(b) Server CPU Utilization

Fig. 2.4 *httpperf* server Throughput and %CPU Utilization on *Xen* virtualized single-core server. The hypervisor and the VMs are pinned to the same core

software bridging to virtualize the NIC. The application throughput loss is the overall effect of virtualization overheads. There is a further drop when the application is hosted on a consolidated server with the VMs sharing the NIC. This is obvious, since for the consolidated server, the NIC is now handling twice the amount of traffic in comparison to that of the virtualized server case. It is interesting to note that the virtualization overheads manifest as extra CPU utilization on the virtualized server [17]. This is observed by the CPU utilization graphs of Figs. 2.3b and 2.4b. Both benchmarks indicate increased CPU activity to support the same application throughput. This imposes response latencies leading to application throughput loss and also usable device bandwidth loss for the VM. The noteworthy side effect of this device bandwidth loss, for a VM, is that it is usable by another VM, which is sharing the device. This is noticed in the throughput graphs of the consolidated server for *netperf* benchmark. It is an encouraging fact for consolidating I/O workloads on

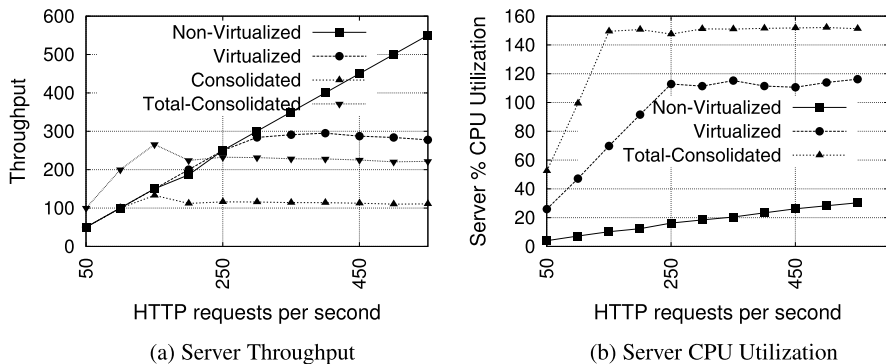


Fig. 2.5 *httpperf* server Throughput and %CPU Utilization on *Vmware-ESXi* virtualized single-core server. The VMs are pinned to a single core, while hypervisor uses all available cores

virtualized servers. However, *httpperf* benchmark performance on the consolidated server is not very impressive and suggests further investigation.

Conducting this experiment on *Vmware* virtualization technology produces similar behavior, which is depicted in Fig. 2.5. The *httpperf* benchmark tests are conducted on an Intel Core2Duo server with two cores. Unlike the case of *Xen*, pinning of ESXi server (the hypervisor) to a CPU is not allowed. Hence, any CPU utilization measurements for the ESXi hypervisor on *Vmware* show utilizations for all CPUs included. This results in %CPU utilization above 100% in the case of multicore systems. *Vmware-ESXi* server implements NIC virtualization using device emulation. It is observed that the overheads of emulation are comparatively quite high in relation to para-virtualization used in *Xen*. Here also, virtualization of NIC results in using up more CPU to support network traffic on a VM when in comparison to a nonvirtualized 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 in comparison to non-virtualized and para-virtualized systems. In this case 70% drop on the maximum sustained throughput is observed in comparison to the throughput achieved in the nonvirtualized environment. This loss is visible even in the consolidated server case. Interestingly, the total network bandwidth used in the case of consolidated VMs on *Vmware-ESXi* 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. It is reasonable to believe that multicores can alleviate the CPU requirement on the consolidated server. On such systems, the CPU requirement of the VMs can be decoupled from that of the VMM by allocating different CPU cores to each of them. Study of *httpperf* benchmark on consolidated server with each VM pinned to a different core, for both *Xen* and *Vmware-ESXi* virtualized server, shows otherwise. Application throughput increase is observed in comparison to single-core consolidated server, but this increase still falls short by 10% of what was achieved for the nonvirtualized server. The reason for this shortcoming is because both VMs sharing the NIC also share the access path that is implemented by the Independent

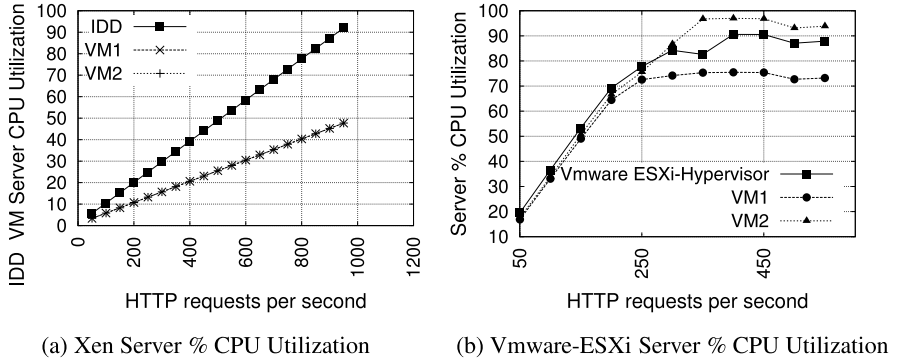


Fig. 2.6 *httpperf* server %CPU Utilization on *Xen* and *Vmware-ESXi* virtualized multicore server. The hypervisor and the VMs are pinned to independent cores

Driver Domain (IDD) in the case of *Xen* and the hypervisor in the case of *Vmware-ESXi* virtualized server. This sharing manifests as serialization and increased CPU utilization of the IDD or the hypervisor, which becomes the bottleneck as the workload increases. Also, this bottleneck restricts the number of VMs that can share the NIC. This is clearly depicted in the graphs of Fig. 2.6. In Fig. 2.6a, it is observed that as the *httpperf* workload is increasing, there is a linear increase in the CPU utilization of the VMs as well as the Xen-IDD hosting the NIC. The CPU utilization of the IDD, however, is much more when compared to the CPU utilization of either of the VMs. This is because the IDD is supporting network streams to both the VMs. As a consequence, it is observed that even though there is spare CPU available to the VMs, they cannot support higher throughput since the IDD has exhausted its CPU resource. This indicates that lack of concurrent device with concurrent access imposes serialization constraints on the device and its access path which limits device sharing scalability on virtualized servers. This behavior is also observed in the case of the *Vmware-ESXi* server as is depicted in Fig. 2.6b. However, as in the case of single-core experiments, the CPU Utilization by the hypervisor and the VMs is significantly much higher in comparison to the *Xen* server for the same benchmark workload. This results in poor performance when compared to para-virtualized devices, but yields more unused device bandwidth. As a result, *Vmware-ESXi* server supports higher scalability for sharing the NIC.

The analysis for multicore virtualized server CPU Utilization indicates that even with the availability of required resources, for each of the VMs and the hypervisor, the device sharing architecture has constraints that impose severe restrictions in usable bandwidth and scalability of device sharing. These constraints are specifically due to serialization of device and its access paths. Hence, it is necessary to rearchitect device virtualization to enable concurrent device access to eliminate the bottlenecks evident in device sharing by the VMs.

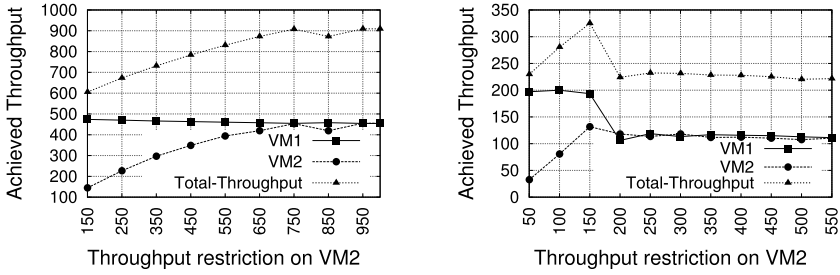
2.3.2 Evaluation of Network QoS Controls

The most noteworthy point of observation of the study in Sect. 2.3.1 is the behavior of each stream of benchmark on the consolidated server. In general, it is observed that there is a further reduction of throughput on the consolidated server in comparison to the single VM on a virtualized server, for both the benchmarks *netperf* and *httperf*, with a marked decrement in the latter case. This indicates the obvious: lack of QoS constraints would lead to severe interference in performance delivered by the device sharing VMs.

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* [11]. Also, as discussed in [17, 24, 31, 32], the existing CPU resource controls are fine-grained enough to deliver desired performance for CPU-based workloads. The problem is with I/O devices. The access to an I/O device is always through the hypervisor or the driver domain 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 hypervisor or the driver domain. Hence, resource allocation controls with regard to the I/O devices are at a higher abstraction level rather than at the device level, unlike in the case of the CPU resource. These controls are effective for the outgoing streams from the server, since packets that overflow are dropped before reaching the NIC. However, for the incoming stream, the control is ineffective since the decision of accepting or rejecting is made after the packet is received by the NIC. Hence, the controls are coarse-grained and affect the way resource usage is controlled and thereby the application performance. 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, thereby causing unpredictable application performance, as is illustrated in the next section.

To understand the effect of software-based QoS controls for network bandwidth sharing, an experimental analysis of *httperf* benchmark on a consolidated server is presented. The consolidated server hosts two VMs, namely VM1 and VM2, that are sharing a NIC. Each VM hosts one *http* server that responds to a *httperf* client. The *httperf* benchmark is chosen for this study because it allows customization of observation time of the experiment. This is necessary since the bandwidth control mechanisms that are available are based on time-sampled averages and hence, need a certain interval of time to affect application throughput. The experiment involves two studies, one is that of best effort sharing where no QoS is imposed on either of the VMs, and in the second case VM1 is allowed to use the available network bandwidth when VM2 is constrained, by imposing specific QoS value based on the desired application throughput. For both studies, each VM is subjected to equal load from the *httperf* clients.

The performance of consolidated server corresponding to the best effort sharing case is presented in Figs. 2.4a and 2.5a. As it is observed from the graphs, the NIC bandwidth sharing is equal in both the virtualization solutions. When no QoS



(a) *Xen-IDD Linux* based QoS controls on VM2 (b) *Vmware-ESXi* QoS controls on VM2

Fig. 2.7 Effect of hypervisor network bandwidth controls on application throughput for consolidated virtualized server hosting two VMs

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, when bandwidth control is enforced on the VM2, while allowing complete available bandwidth to the VM1, the expected behavior is to see improved throughput for the unconstrained VM1. This is to say, VM1 performance is expected to be better in comparison to the best effort case. Figure 2.7 demonstrates that imposing QoS controls on VM2 does not translate to extra bandwidth availability for the other, unconstrained VM. The reasons for this behavior are a multitude. The most significant ones being the virtualization overhead in terms of the CPU resource required by the VMM or the hosting VM to support I/O workload, serialization of the resource and its access path, lack of control on the device for the VM specific incoming network stream, and lastly, higher priority to the incoming stream over the outgoing stream at the device. All these lead to unpredictable application performance inspite of applying appropriate QoS controls. Also, it is interesting to note that the variation in performance is dependent on the nature of the consolidated workloads. This performance variation affects all the consolidated workloads and makes the application guarantee weak. On multi-core servers hosting many consolidated workloads of a datacenter, indeterminate performance is definitely not acceptable. Also, since virtual device is an abstraction supported in software, device usage controls are coarse grained and hence ineffective. This could lead to an easy denial of service attack on a consolidated server with shared devices.

The bandwidth controls enforced are 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 *httpperf* request rates, without performance loss, is measured. These bandwidth measurements are used to apply control on the outgoing traffic from VM2. Currently, the available controls allow constraints only on the outgoing traffic. On the incoming traffic, ideally the control should be applied at the device so that any packet causing overflow is dropped before reception. Such controls are not available at present. Instead, in *Xen*, at least one can use the *netfilter module's* stream-based controls after receiving the packet. This does not serve the purpose, because by receiving a packet that could potentially be dropped

later, the device bandwidth is anyway wasted. Hence, the study involves using only the outgoing traffic controls for the constrained VM.

The selection of different range of workloads, for each of the virtualized server, is based on the maximum throughput that each can support in a consolidated server environment. For each QoS control, the maximum throughput achieved, without loss, by each of the VM, is plotted in the graphs of Figs. 2.7a and b. In these figures, the x -axis represents the *httperf* request rate based on which the network bandwidth control was applied on the VM2, and the y -axis represents the application throughput achieved by each of the VMs. In the case of *Xen*, *Linux tc* utility of the *netfilter* module [34] is used to establish appropriate bandwidth controls. Specifically, each traffic stream from the VMs is defined using *htb* class with *tbpf* queue discipline with the desired bandwidth control. Each queue is configured with a burst value to support a maximum of 10 extra packets. In the case of *Vmware-ESXi* server, the *Veam Monitor* controls for network bandwidth are used and populated with the same QoS controls as is done for the *Xen* server.

Based on the behavior of the benchmarks, following bottlenecks are identified for sharing network I/O device across multiple VMs on *Xen* or *Vmware-ESXi* virtualized server.

- Virtualization increases the device utilization overheads, which leads to increased CPU utilization of the hypervisor or the IDD hosting the device.
- Virtualization overheads cause loss of device bandwidth utilization from inside a VM. Consolidation improves the overall 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 themselves become the bottlenecks for sharing the device.
- QoS features for regulating incoming and outgoing traffic are currently implemented in the software stack. Uncontrolled incoming traffic at the device, to a VM that is 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. This could potentially cause a denial of service attack on the VMs sharing the device.

Based on the above study, a device virtualization architecture is proposed and described in the following sections. The proposal is an extension to I/O virtualization architecture, beyond what is recommended by the PCI-SIG IOV specification [21]. The PCI-SIG IOV specification defines the rudiments for making I/O devices virtualization aware. On the multicore 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. High-speed network devices, like 10-Gbps NICs, are available 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 controls, without compromising on the device utilization. The architecture is designed to enable native access of I/O devices to the VMs 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 device access and eliminating hypervisor/IDD from the path of device access.
- Enable fine-grained resource usage controls at the device.

In the remaining part of the chapter, we bring out the need for extending I/O device virtualization architecture in Sect. 2.4. Section 2.5 highlights the issues in sharing of the I/O device and its access path in prevalent virtualization architectures leading to a detailed description of the proposed architecture to overcome the bottlenecks. *Xen* virtualization architecture is taken as the reference model for the proposed architecture. In the subsequent part of the section, a complete description of the network packet work-flow for the proposed architecture is presented. These work-flows form a basis for generating the LQN model that is used in the simulation studies for architecture evaluation described in Sect. 2.6. A brief description of the LQN model generation and detailed presentation of simulation results is covered in Sect. 2.7. Finally, in Sect. 2.8 the chapter conclusion highlights on the benefits of the architecture.

2.4 Review of I/O Virtualization Techniques

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 as in *Vmware* [15] and para-virtualization as in *Xen* [11]. In full system virtualization complete hardware is replicated virtually. Instruction emulation is used to support multiple architectures. The advantage of full system virtualization is that it enables unmodified Guest operating systems (GuestOS) to execute on the VM. Since it adopts instruction emulation, it tends to have high performance overheads as observed in the experimental studies described earlier. In Para-virtualization the GuestOS is also modified suitably to run concurrently with other VMs on the same hardware. Hence, it is more efficient and offers lower performance overheads. In either case, system virtualization is enabled by a layer called the virtual machine monitor (VMM), also known as the hypervisor, that provides the resource management functionality across multiple VMs. 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 [27]. A dedicated software entity, called the I/O domain, is used to perform physical device management [9, 12]. 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 [14, 19, 20, 22].

Recent publications on concurrent direct network access (CDNA) [23] and scalable self-virtualizing network interface [16] are similar to the proposed work in the sense that they explore the I/O virtualization issues on the multicore platforms and provision for concurrent device access. However, the scalable self-virtualizing interface 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 authors do not detail how the address translation issues are handled, particularly in the case of virtualized environments. CDNA is architecturally closer to our architecture since it addresses concurrent device access by multiple VMs. CDNA relies on per VM Receive (Rx) and Transmit (Tx) ring buffers to manage VM specific network data. The VMM handles the virtual interrupts, and the *Xen* implementation still uses IDD to share the I/O device. Also, authors do not address the performance interference due to uncontrolled data reception by the device nor do they discuss the need for addressing the QoS controls at the device level.

The proposed architecture addresses these and suggests pushing the basic constructs to assign QoS attributes like required bandwidth and priority into the device to get fine-grained control on interference effects. Also, the proposed architecture has its basis in *exokernel's* [6] philosophy of separating device management from protection. In *exokernel*, the idea was to extend native device access to applications with the *exokernel* providing the protection. In the proposed approach, the extension of native device access is with the VM, the protection being managed by the VMM and the device collectively. A VM is assumed to be running the traditional GuestOS without any modifications with native device drivers. This is a strong point in support of legacy environments without any need for code modification. 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 [18], Neterion [25], NetXen [26], Solarflare [33], etc. Apart from these, the Crossbow [28] suite from SUN Microsystems talks about this kind of resource provisioning. However, Crossbow is a software stack over a standard IOV compliant hardware. The results published using any of these products are exciting in terms of the performance achieved. These devices when used within the prevalent virtualization technologies need to still address the issue of provisioning QoS controls on the device. Lack of such controls, as illustrated by the previously described experimental studies, cause performance degradation and interference that is dependent on the workloads sharing the device.

2.5 Enhancement to I/O Virtualization Architecture

The analysis of prevalent commodity virtualization technologies in Sect. 2.3 clearly highlights the issues that need to be addressed while sharing I/O devices across independent VMs on multicore virtualized servers. It is also observed that while para-virtualization offers better performance for the application, emulation is an alternative for improved consolidation. The goals are seemingly orthogonal since current technologies build over virtualization unaware I/O devices. The proposed architecture takes a consolidated perspective of merging these two goals, that of ensuring application performance without losing out on the device utilization by taking advantage of virtualization aware I/O devices and rearchitecting the end-to-end virtualization architecture to deliver the benefits. In order to understand the benefits of the proposed architecture, the *Xen*-based para-virtualization architecture for I/O devices is taken as the reference model. In the existing *Xen* virtualization architecture, analysis of the network packet work-flow highlights following bottlenecks:

- Since the NIC 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 *Xen*-IDD is the critical section for all the VMs sharing the device. IDD incurs processing overheads for every network operation executed on behalf of each VM. Current IDD implementations do not have any hooks for controlling the overheads on 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. Service overheads of this stage-wise data movement cause drop in effective utilized device bandwidth. In multicore servers with scarce I/O devices, this would mean having high-bandwidth under-utilized devices and low-throughput applications on the consolidated server.

To overcome the above-listed drawbacks, the proposed architecture enhances I/O device virtualization to enable separation of device management from device access. 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 recognizes the destination VM of an incoming packet by the interrupt raised by the device and forwards it to the appropriate VM. The VM then processes the packet as it would do so in the case of nonvirtualized environment. Thus, device access and scheduling of device communication are managed by the VM that is using it. The identity for access is managed by the VMM. This eliminates the intermediary VMM/IDD on the device access path and reduces I/O service time, which improves the application performance on virtualized servers and also the usable device bandwidth which results in improved consolidation. In the following subsections we describe the NIC I/O virtualization architecture, keeping the above goals in mind, and suggest how the system software layers of the VMM and the GuestOS inside the VM should use the NIC hardware that is enabled for QoS-based concurrent device access.

2.5.1 Proposed I/O Virtualization Architecture Description

Figure 2.8 gives a block schematic of the proposed I/O virtualization architecture. The picture depicts a NIC card that can be housed within a multicore 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 $m \leq n$, along with device registers, forms the virtual-NICs (vNICs). The device memory is reconfigurable, i.e., dynamically partitionable, and the VM's QoS requirements drive the sizing of the memory partition of a vNIC. 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 support adaptive QoS specifications. The NIC identifies the destination VM of an arriving packet, based on the logical device address associated with it. A simple implementation is to allow a single physical NIC to support multiple MAC address associations. Each MAC address then represents a vNIC, and a vNIC request is identified by generating a message-signaled interrupt (MSI). The number of MAC addresses and interrupts supported by the controller restricts the number of vNICs 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 a vNIC. The VMM can aid in setting up the appropriate hosting connections based on the requested QoS requirements. The architecture can be realized with the following enhancements:

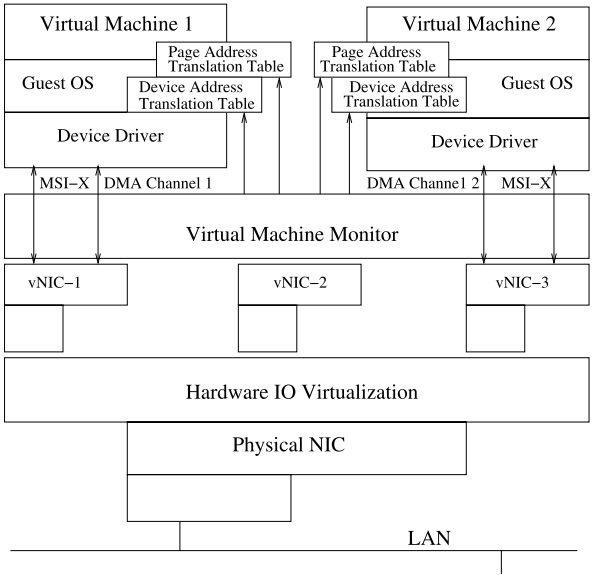


Fig. 2.8 NIC architecture supporting independent reconfigurable virtual-NICs

Virtual-NIC

In order to define vNIC, the physical device should support timesharing 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 Fig. 2.8. 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. This virtual interface is called the vNIC which forms a restricted address space on the device for the VM to use and remains in possession of the VM until it is active or relinquishes the device. The VMM sets up the device page translation table, mapping the physical device address of the vNIC into the virtual memory of the importing VM, during the vNIC creation and initialization. The device page translation table is given read-only access to the VM and hence forms a significant security provisioning on the device. This prohibits any corrupt device driver of the VM GuestOs to affect other VMs sharing the device or the VMM itself. Also, for high-speed NIC devices, the partitionable memory of the vNIC is useful in setting up large receive and segment offload capabilities specific to each vNIC and thus customizes the sizing of each vNIC based on the QoS requirements of the VM.

Accessing Virtual-NIC

To access the vNIC, the native device driver hosted inside the VM replaces the IDD layer. This device driver manipulates the restricted device address space which is exported through the vNIC interface by the VMM. The VMM identifies and forwards the device interrupt to the destination VM. The GuestOS of the VM handles the I/O access and thus directly accounts for the resource usage it incurs. This eliminates the performance interference when the IDD handles multiple VM requests to a shared device. Also, direct access of vNIC to the VM reduces the service time on the I/O accesses. This results in better bandwidth utilization. With the vNIC interface, data transfer is handled by the VM. The VM sets up the Rx/Tx descriptor rings within its address space and makes a request to the VMM for initializing the I/O page translation table during bootup. The device driver uses this table along with the device address translation 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. With appropriate logic on the NIC card, QoS-specific service level agreements (SLAs) can be easily implemented on the device that translates to bandwidth restrictions and VM-based processing priority. The key is being able to identify the

incoming packet to the corresponding VM. This is done by the NIC based on the packet's associated logical device address. The NIC controller decides on whether to accept or reject the incoming packet based on the bandwidth specification or the current free memory available with the destination vNIC of the packet. 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.

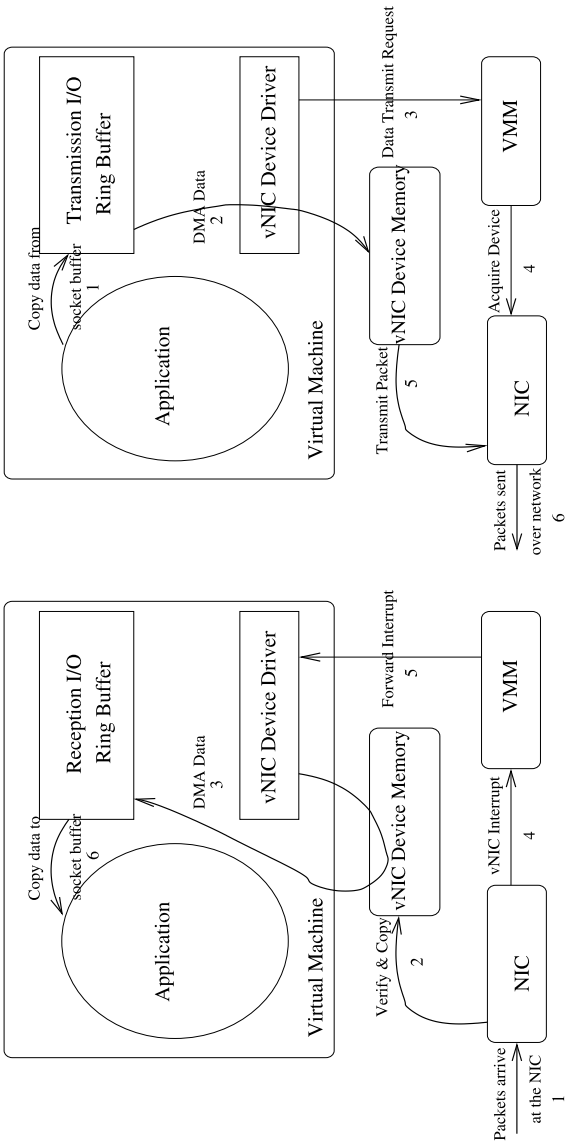
Security and Virtual-NIC

Each vNIC is carved out as a device partition, based on the device requirement specification of the VM. By using appropriate microarchitecture and hardware constructs it can be ensured that a VM does not monopolize device usage and cause denial of service attack to other VMs sharing the device. The architecture allows for unmodified GuestOS on a VM. Hence the security is verified and built outside the VM, i.e., within the VMM. Allowing native device driver within the VM for the vNIC not only enhances the performance but also allows for easy trapping of the device driver errors by the VMM. This enables for building robust recovery mechanisms for the VM. The model also eliminates sharing of the device access path by allowing direct access to the vNIC by the VM and thereby eliminates the associated failures [35].

With these constructs, the virtualized NIC is now enabled for carving out secure, customized vNICs for each VM, based on its QoS requirements, and supports native device access to the GuestOS of the VM.

2.5.2 Network Packet Work-Flow Using the Virtualized I/O Architecture

With the proposed I/O device virtualization architecture, each VM gets safe, direct access to the shared I/O device without having to route the request through the IDD. Only the device interrupts are routed through the VMM. In Figs. 2.9a and b, 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 vNIC's portion of the device memory and issues DMA request to the destination VM based on the vNIC's priority. On completion of the DMA request, the device raises an interrupt. The VMM intercepts the interrupt, determines the destination VM, and forwards the interrupt to the VM. The VM's device driver then receives the data from the VM specific device descriptor rings as it would do in the case of nonvirtualized server. In the case of transmission, the device driver that is resident in GuestOS of the VM does a DMA transfer of the data directly into the vNIC's mapped memory and sets the appropriate registers to



(a) Packet reception work-flow

(b) Packet transmission work-flow

Fig. 2.9 Workflow of network I/O communication with improvised I/O device virtualization architecture

initiate data transmission. The NIC transmits this data based on the vNICs properties like speed, bandwidth, and priority. It may be worth noting here that the code changes to support this architecture in the existing implementation will be minimal. Each VM can use the native device driver for its vNIC. This device driver is the standard device driver for the IOV compliant devices with the only difference that it can only access restricted device address. 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.

2.6 Evaluation of Proposed Architecture

Since the architecture involves the design of a new NIC and a change in both VMM and the device handling code inside the VM's GuestOS, evaluation of the architecture is carried out using simulation based on LQN model of the architecture. In LQN models, functional components of the architecture workflow are represented as server entries. Service of each entry is rendered on a resource. End-to-end workflow is enacted using entry interactions. The LQN models capture the contention at the resource or software component using service queues. The reason for choosing LQN-based modeling is twofold. First, there is a lack of appropriate system simulation tools that allow incorporating design of new hardware along with VMM and GuestOS changes. Second, LQN models are intuitive queuing models that enable capturing of the device and software contention and associated serialization 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 are effective in capturing the causes of bottlenecks in the access path. For complete details on general description of LQN modeling and simulation, the reader may refer to [3–5].

2.6.1 LQN Model for the Proposed Architecture

LQN models can be generated based on the network packet receive and transmit workflows, manually, using the LQNDEF [3] software developed at the RADS laboratory of *Carleton University*. In the chapter, results generated for the LQN model corresponding to the *httperf* benchmark are presented for analysis, since the bottleneck issues are prominent for this benchmark. For complete details on the generation of the LQN models for the *httperf* benchmark and validation of the models against experimental data, readers may refer to [29, 30]. Three assumptions are made while generating the LQN models used for this analysis, namely:

- The service times established at each of the entries constituting the LQN model are 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 enable measurement of the model throughput in terms of the number of satisfied requests. The model validation results demonstrate that there is no significant loss or gain ($<1\%$) of throughput because of this.

- The experimental results for *httperf* benchmark illustrated in Sect. 2.3 are carried out with varying request rates for a single specified file. In this mode of execution, the file that is fetched as a reply to each of the *http* request, remains constant. Hence the measured service time to process each request remains constant. Also, for the chosen mode of execution of the *httperf* benchmark, the arrival request rate is observed to be uniform. Hence, the service times and arrival rates populated on the LQN model are modeled as deterministic.
- The service time for all device activities that are assumed to be executed in hardware, in the proposed architecture and modeled as separate entities in the LQN model, is set to be significantly low (10^{-10} seconds). For the rest of the software entries, the service times are derived based on the measurements made for the nonvirtualized servers. This is justified since the proposed architecture gives native access to the device from within the VM which is assumed to be running the same GuestOS as is used for the nonvirtualized server.

In general it is observed that the maximum throughput observed using the LQN model is 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 recorded by the profiler as the service time. In the real system, to match the difference between the device speed and CPU speed, appropriate memory buffers (TCP transmit and receive buffers of *Linux* kernel) are maintained. The sizing of these buffers affects the observed application throughput. Observed throughputs are higher for larger buffer sizes. This trend is maintained to the point until the device can handle the rate of network traffic. Once device saturation occurs, the failure behavior usually results in a sudden drop in application throughput. While setting up the LQN model, the maximum permissible default buffer size was used in the simulator (which is more than three times than what was set on the experimental system). This is normally the adopted practice since in throughput studies the interest is to understand the limits of the model for those service times that make the contention predominant. This gives an idea on the upper bound of application throughput on a system with maximum possible resources for the service times possible within the desired architecture. The basic 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 nonvirtualized server, the results observed using simulations are contradictory. This is because in the simulation environment, the buffer sizes used were much larger than the experimental system. Hence, to make the comparison fair, normalization of simulation results for existing architecture is carried out. To normalize, the LQN model of existing *Xen* architecture is built, and simulation results are generated.

These results are verified and validated for correctness with that of observed experimental results. After this, all comparisons for the proposed architecture are made using the simulation results of the existing architecture rather than the experimental results.

2.7 Simulation and Results

The proposed architecture is evaluated using the *parasrvn* simulator of the *LQNS* software package [3]. The architecture is evaluated for multicore virtualized servers since the illustrated device sharing dynamics are expected to be pertinent to such systems. The LQN model built for this study consists of one VMM and two VMs, and each is pinned to an independent core. In order to compare the performance of the proposed end-to-end architecture within the simulation environment, validation of the LQN model for the existing *Xen* architecture for a multicore server is carried out. Figure 2.10 depicts the results of achievable throughput and server CPU utilization for a multicore *Xen* server with two VMs consolidated. The throughput graph for both the VMs is similar and appears overlapped in the chart. As it can be noted from Figs. 2.10a and b, in a multicore 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 950 requests/s as against 450 requests/s in the case of single core. But, for the maximum throughput, it is observed that the *Xen-IDD*, which is hosting the NIC of the server, the CPU utilization saturates. This indicates that further increase in application throughput is impossible since the processor core serving the *Xen-IDD* has no computing power left. Figure 2.11 shows these statistics for a similar situation but with the proposed I/O virtualization architecture. As one can observe from Fig. 2.11a, the maximum throughput achievable now per VM increases to 1500 requests/s. This is an increase of application throughput by about 60%. The total throughput achievable at the NIC, derived from consolidating the throughput of both the VMs, also increases by 60% in comparison to what was achieved on the existing *Xen* architecture.

Also, from Fig. 2.11b it is observed that the CPU utilization of the *IDD* or the hypervisor has considerably reduced and remains bounded by an upper limit. The reason for this behavior is that the NIC is now handling the identity of the packet destination. Also, in the existing model, bridging software, which routes the packets to a VM and has a substantial overhead, is eliminated in the proposed architecture. The effect is a reduction in the processing time that the *IDD* spends on behalf of each VM. It is also noticed that since the VMM is now spending almost constant time on I/O requests on behalf of the VMs, there is an elimination of performance interference due to varying workloads. This improves the scalability of sharing the device across VMs. With the proposed architecture, each VM is now accountable for all the resource consumption, thereby leading to better QoS controls.

The next evaluation of the proposed architecture is for QoS controls on the network bandwidth. Since the architecture is implemented using LQN model, certain

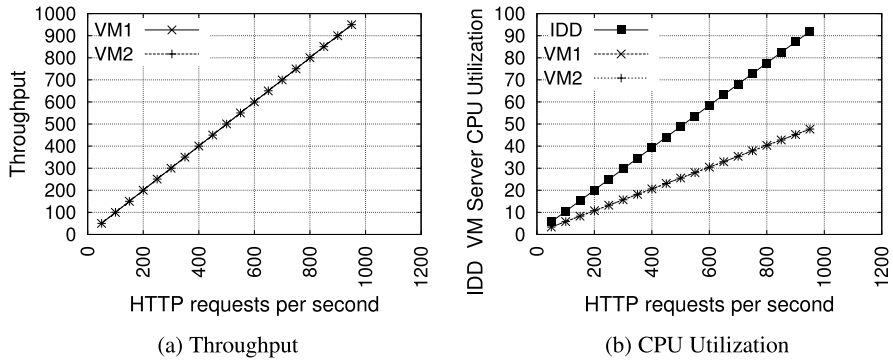


Fig. 2.10 Maximum throughput achievable per *httperf* stream and CPU utilization for existing *Xen* architecture on a multicore server hosting two VMs, each servicing one of the *httperf* stream. The IDD, VM1, and VM2 are pinned to independent cores

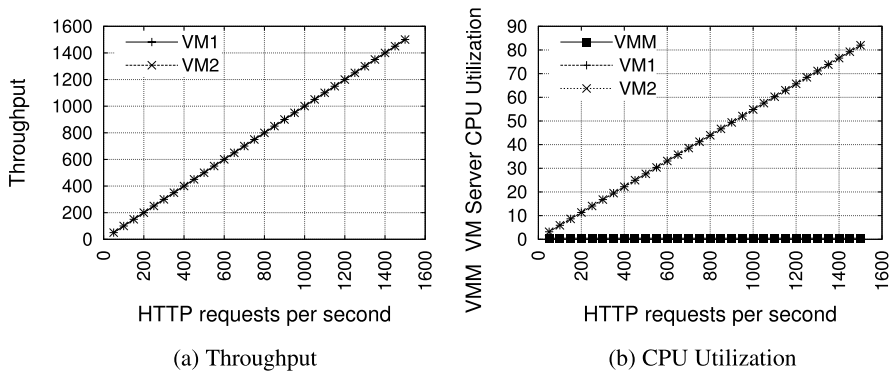


Fig. 2.11 Maximum achievable throughput and CPU utilization charts for a multicore virtualized server incorporating the proposed I/O virtualization architecture and hosting two VMs, pinned to different cores, each servicing one *httperf* stream

modeling assumptions are made to simulate the network bandwidth controls as implemented in the *netfilter* module of *Linux*. LQN model is basically a queuing model wherein 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 represents 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 model 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 resource time used for servicing the request by the

entry of LQN model, and think time is used to model bandwidth restriction. For example, to model 250 requests/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. The next parameter to model is the burst parameter of the bandwidth control mechanism in *Linux netfilter* module. In *Linux netfilter* module, once the bandwidth limit is reached, packet loss occurs. The bandwidth control mechanism also has a burst parameter that allows for some extra packet delivery on the channel, over and above that of imposed bandwidth restriction. By setting the burst rate sufficiently low, equivalent to 10 packets, which is also the minimum that is permissible, it is ensured that the bandwidth control on the constrained channel is tight. The *HTML* page that is requested in the experiments requires fourteen packets to complete a successful request. Since there is no feature in LQN model to associate the burst parameter of *netfilter*, the QoS experiments were carried out by setting the burst rate to 10 packets. This ensures that for the request that exceeds the configured bandwidth, control fails, and the throughput reported takes into account the desired behavior. Thus, think time setting in LQN model is more restrictive than the *netfilter*. However, since the think time value is based on the deterministic request rate parameter that defines the bandwidth constraint, it still produces equivalent results, and this has been validated against observed experimental values [29].

The following graphs in Fig. 2.12 depict the effect of not imposing (Fig. 2.12a) and imposing network bandwidth QoS controls on the incoming stream of VM2 (Fig. 2.12b), in the proposed architecture. The simulations are conducted on a single core server to keep the achievable throughput range within reasonable simulation time. As it can be observed from the graphs of Fig. 2.12a, for the best effort service, the maximum throughput, without loss, achieved by either of the VMs on the consolidated server is equal, indicating a fair share of the resource. The graphs of the Fig. 2.12b 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 the figure, VM2 is constrained to allow requests starting from 150 requests/second to 950 requests/second, and VM1 is unconstrained. Since

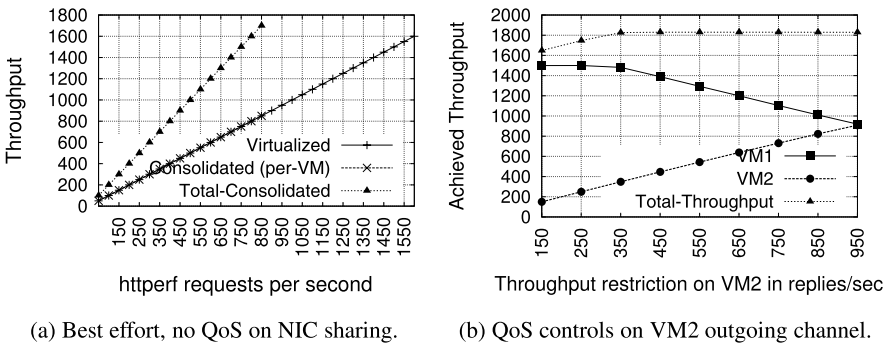


Fig. 2.12 Throughput achieved before and after imposing QoS controls on VM2 of the proposed architecture

the NIC is discarding requests to VM2 that are above the specified request rate, VM1 can use the available bandwidth, and hence higher throughput (1500 replies/sec) on VM1 is achievable. As the bandwidth control on VM2 is relaxed, it is noticed that the throughput graphs start converging toward each other and finally merge to that of the best effort case. The bandwidth control on the incoming stream also works to our advantage on the *http* traffic, because by discarding the request at the device itself, the server and hence the associated resources are spared to respond on requests that will eventually be dropped because of bandwidth controls. This control on the device also acts as a strong deterrent for any denial of service type of attacks. 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 priority and bandwidth control on the outgoing channel of the constrained VM. The outgoing channel constraints are easily achievable by using existing mechanisms such as those available in the *netfilter* module of *Linux* [34]. The important point to note here is that with faster and higher-bandwidth NIC devices, judicious use of large receive and segment offload buffers can lead to higher device utilization without compromising the VM's performance.

2.8 Conclusion

In this chapter, we described how the lack of virtualization awareness in I/O devices can lead to latency overheads on the I/O path and also cause security vulnerabilities. In addition to this, the intermixing of device management and data protection issues further increases the latency. This results in reducing the effective usable bandwidth of the device. Also, lack of appropriate device-sharing control mechanisms, at the device level, leads to loss in bandwidth, causes performance interference on the device sharing VMs, and makes the virtualization software the most vulnerable component of the consolidated server. To address these issues, I/O device virtualization architecture is proposed. The architecture is an extension to the PCI-SIG IOV specification. The architecture evaluation is 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 performing any software optimization on the I/O path. The proposed architecture also improves the security and scalability of VMs sharing the NIC. It is 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 is done for para-virtualized systems like *Xen*, it is reasonable to expect that the ideas presented would benefit fully virtualized systems like *Vmware* since the architecture enables elimination of the common software entity by providing native device access to the GuestOS of the VM.

Acknowledgements Credits for this work are due to all those unknown reviewers who have meticulously pointed out deficiencies and improvements over several rounds of reviews and also to the summer interns who have enthusiastically carried out the numerous experimental work that helped validate the simulation results.

Appendix

Layered Queuing Network (LQN) models are the queuing models designed to capture the interdependencies in layered systems. The complete system is described by a set of operations carried out over a set of resources. Every operation requires one or more resources for execution. The LQN model defines an architectural and resource context for each operation. The architectural context defines the initiating event for the operation (execution trigger), when the execution should begin (execution timing) and when it should complete (completion trigger). Based on the semantics of the architectural context, the operation uses resources to carry out its activities, which is defined by its resource context. A resource can be a software entity or a hardware unit involved in actual execution of the operation. Each resource is associated with a queue with a discipline that enforces the order of resource use by the tasks. In layered systems, execution of an activity is carried out by a structured order of operations over resources organized in different layers. An LQN model is necessarily an acyclic graph of all possible sequences of requests to avoid the issue of resource deadlocks. LQNs are very intuitive in capturing resource contentions and thereby the performance implications on a layered system. These models are quite common in practice for modeling software system performance.

The LQN models used in this chapter to evaluate I/O virtualization architecture for the *httperf* benchmark are generated using the software developed at the RADS Laboratory of Carleton University. Complete details of the software, tools, and the associated documentation can be found on their website [3].

A short description of the LQN models generated for the proposed I/O virtualization architecture and *Xen* is provided here. The I/O virtualization issues are prominent for the *httperf* benchmark, and hence LQN models that capture the end-to-end architecture are generated for analyzing the issues. The diagrams in Figs. 2.13 and 2.14 depict the LQN models generated for a consolidated *Xen* server and the proposed I/O virtualization architecture, hosting two VMs. The model has two *httperf* streams accessing *http* servers hosted on different VMs. The model captures the scenario for a multicore system. In these models, each rectangular box represents the conceptual functional entity that is active in the receive or the transmit path of the network packet workflows depicted in Fig. 2.9, to complete one *httperf* request-reply sequence. To make the LQN model simpler, a few assumptions are made:

1. While in reality every *http* request is broken into a sequence of packets that are passed through various layers of OS, on an LQN model it is captured as a single service request. This allows for throughput measurements on the model in terms of satisfied *http* requests. This is the unit of measurement for the *httperf* benchmark. By aggregating contention issues from packet level to request level, the throughput measurements tend to be optimistic than what is observed in actual experiments.
2. The service time associated with the transmit/receive operation is consolidated to represent the sending of all the packets composing the *http* request. Because of this assumption, the results of the simulation tend to give upper bounds on

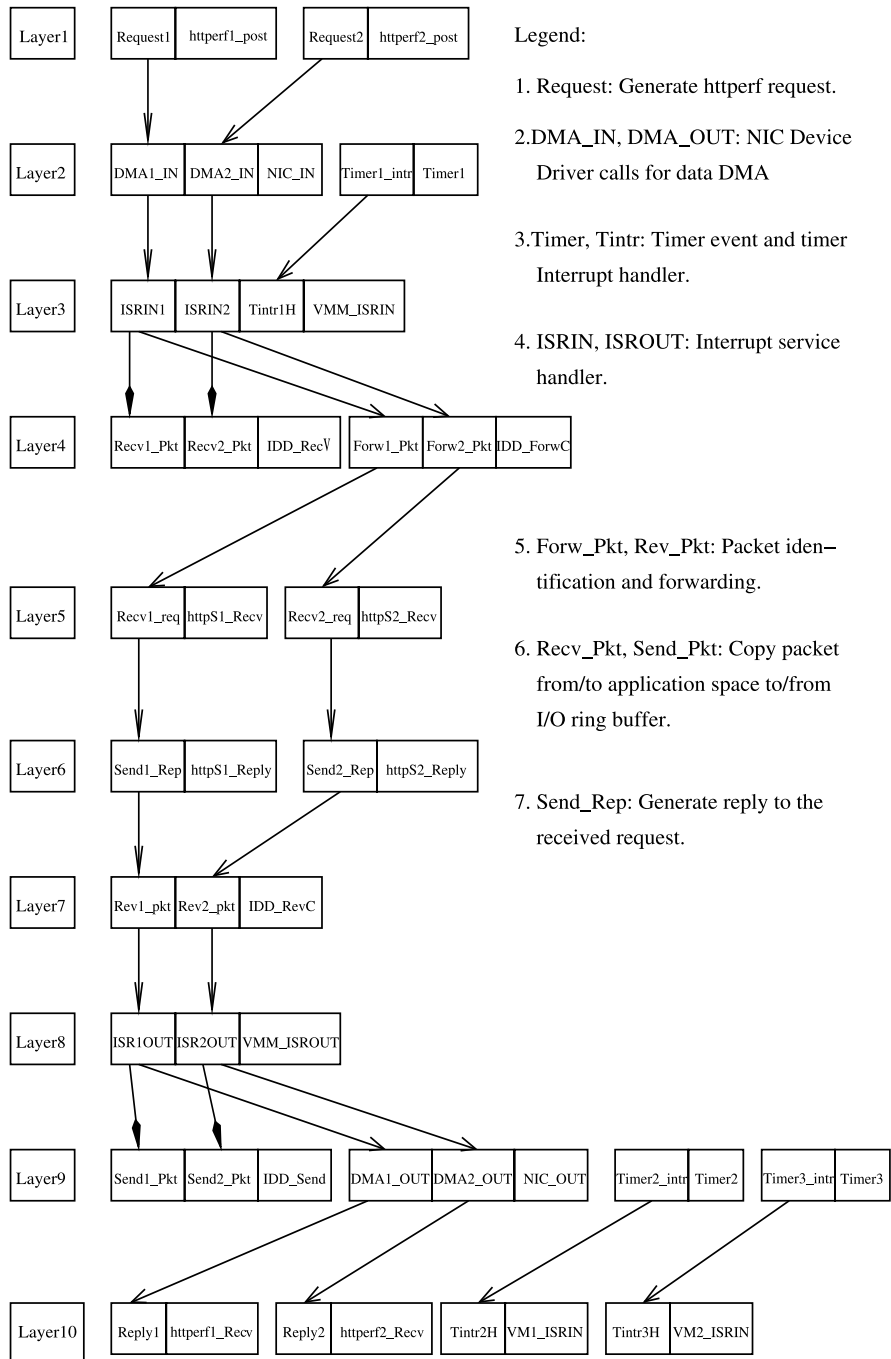


Fig. 2.13 Layered Queuing Network Model for end-to-end *httpperf* benchmark on Xen server

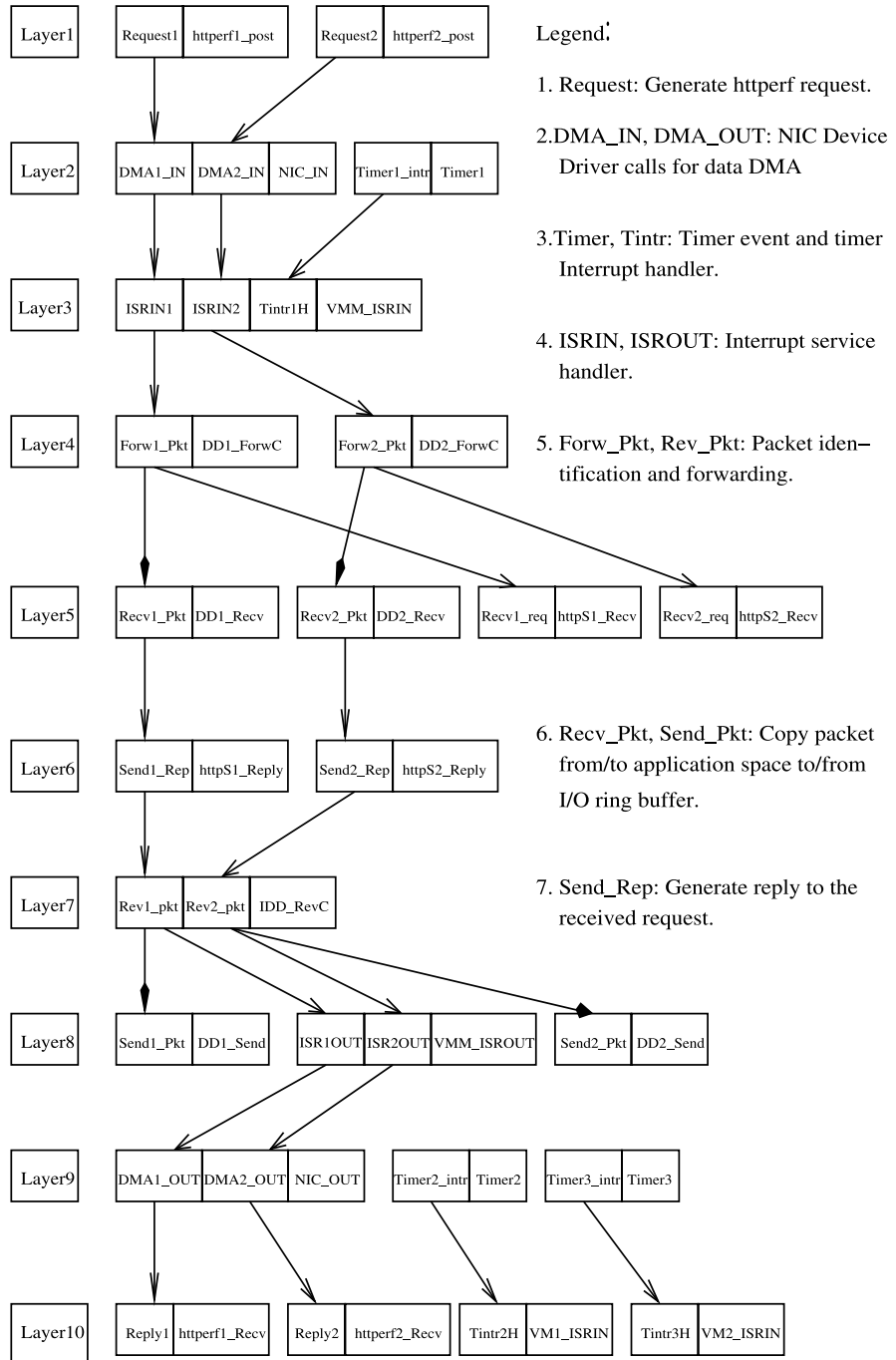


Fig. 2.14 Layered Queuing Network Model for end-to-end *httpperf* benchmark on proposed I/O virtualized server

the achievable throughput when compared to actual implementation. But the deviation is well within 10% of the observed values, as reported in [29, 30]. This makes LQN models very useful in evaluating end-to-end architectures.

3. One element that is incorporated in the LQN model and not shown in the workflow is the system timer interrupt using the server element “Timer.” This element is introduced in the LQN to account for the queuing delays accrued, while the OS is handling timer interrupts. For generating the service time of the interrupt handler, a significantly small delay is used. This value is currently set randomly for want of standard tools to profile kernel procedures.
4. All entries in the LQN model that represent hardware functions are set with a significantly small delay as the service time.

Further details on generating of the LQN models and validating the models against experimental data for this benchmark are discussed in [29, 30].

References

1. Goldberg, R.P.: Survey of virtual machine research. *IEEE Comput.* **7**(6), 34–45 (1974)
2. Jones, R.A.: Netperf: a network performance benchmark revision 2.0. Technical Report, Information Networks Division, Hewlett-Packard Company (1993). Available online: <http://ci.nii.ac.jp/naid/10000088072/en/>. Cited 30 April 2010
3. RADS Carleton Univ.: Layered Queueing Network Solver software package (1995). Available online: <http://www.sce.carleton.ca/rads/lqns>. Cited 30 April 2010
4. Rolia, J.A., Sevcik, K.C.: The method of layers. *IEEE Trans. Softw. Eng.* **21**(8), 689–700 (1995)
5. Woodside, C.M., Neilson, J.E., Petriu, D.C., Majumdar, S.: The stochastic rendezvous network model for performance of synchronous client-server-like distributed software. *IEEE Trans. Comput.* **44**(1), 20–34 (1995)
6. Kaashoek, M.F., et al.: Application performance and flexibility on exokernel systems. In: 16th ACM SOSP, pp. 52–65 (1997)
7. Verghese, B., Gupta, A., Rosenblum, M.: Performance isolation: sharing and isolation in shared-memory multiprocessors. *ACM SIGPLAN Not.* **19**, 181–192 (1998)
8. Mosberger, D., Jin, T.: httpperf: a tool for measuring web server performance. In: ACM Workshop on Internet Server Performance, pp. 59–67 (1998)
9. Sugerman, J., Venkatachalam, G., Lim, B.: Virtualizing I/O devices on VMware workstation’s hosted virtual machine monitor. In: Proceedings of the USENIX Annual Technical Conference, pp. 1–14 (2001)
10. Welsh, M., Culler, D.: Virtualization considered harmful OS design directions for well-conditioned services. In: Hot Topics in OS 8th Workshop, pp. 139–144 (2001)
11. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: 19th ACM SIGOPS, pp. 164–177 (2003)
12. Fraser, K., Hand, S., Neugebauer, R., Pratt, I., Warden, A., Williamson, M.: Safe hardware access with the Xen virtual machine monitor. In: 1st Workshop on OASIS (2004)
13. The Globus Resource Specification Language RSL v1.0 (2004). Available online: http://www-fp.globus.org/gram/rsl_spec1.html. Cited 30 April 2010
14. Menon, Santos, J.R., Turner, Y., Janakiraman, G.J., Zwaenepoel, W.: Diagnosing performance overheads in the Xen virtual machine environment. In: Proceedings of the ACM/USENIX Conference on Virtual Execution Environments, pp. 13–23 (2005)
15. VMware (2005) VMware ESX Server 2—architecture and performance implications (2005). Available online: http://www.vmware.com/pdf/esx2_performance_implications.pdf. Cited 30 April 2010

16. Raj, H., Schwan, K.: Implementing a scalable selfvirtualizing network interface on a multi-core platform. In: Workshop on the Interaction Between Operating Systems and Computer Architecture (2005)
17. Gupta, D., Cherkasova, L., Gardner, R., Vahdat, A.: Enforcing performance isolation across virtual machines in Xen. *Lect. Notes Comput. Sci.* **4290**, 342–362 (2006)
18. Intel Virtualization Technology for Directed-I/O (2006). Available online: www.intel.com/technology/itj/2006/v10i3/2-io/7-conclusion.htm. Cited 30 April 2010
19. Liu, J., Huang, W., Abali, B., Panda, D.K.: High performance VMMbypass I/O in virtual machines. In: Proceedings of the USENIX Annual Technical Conference, pp. 3–3 (2006)
20. Menon, Cox, A.L., Zwaenepoel, W.: Optimizing network virtualization in Xen. In: Proceedings of the USENIX Annual Technical Conference, pp. 2–2 (2006)
21. PCI-SIG IOV Specification (2006). Available online: <http://www.pcisig.com/specifications/iov>. Cited 30 April 2010
22. Santos, J.R., Janakiraman, G., Turner, Y., Pratt, I.: Netchannel 2: optimizing network performance. In: Xen Summit Talk (2007)
23. 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, pp. 306–317 (2007)
24. Nesbit, K.J., Moreto, M., Cazorla, F.J., Ramirez, A., Valero, M., Smith, J.E.: Multicore resource management. *IEEE Micro* **28**(3), 6–16 (2008). Special Issue on Interaction of Computer Architecture and Operating System in the Manycore Era
25. Neterion (2008). Available online: <http://www.neterion.com/>. Cited 30 April 2010
26. Netxen (2008). Available online: <http://www.netxen.com/>. Cited 30 April 2010
27. Rixner, S.: Breaking the performance barrier: shared I/O in virtualization platforms has come a long way but performance concerns remain. *ACM Queue* **6**(1), 36 (2008)
28. Sun Microsystems: CrossBow Network Virtualization and Resource Control (2008). Available online: http://www.opensolaris.org/os/community/networking/crossbow_sunlabs_ext.pdf. Cited 30 April 2010
29. Lakshmi, J., Nandy, S.K.: Modeling Architecture-OS interactions using layered queuing network models. In: International Conference Proceedings of HPC Asia, pp. 382–389 (2009)
30. Lakshmi, J., Nandy, S.K.: I/O device virtualization in multi-core era, a QoS perspective. In: Workshop on Grids, Clouds and Virtualization, Conference on Grids and Pervasive Computing, pp. 128–135 (2009)
31. Kim, H., Lim, H., Jeong, J., Jo, H., Lee, J.: Task-aware virtual machine scheduling for I/O performance. In: Proceedings of ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pp. 101–110 (2009)
32. Weng, C., Wang, Z., Li, M., Lu, X.: The hybrid scheduling framework for virtual machine systems. In: Proceedings of ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pp. 111–120 (2009)
33. Solarflare Communications (2009). Available online: <http://www.solarflare.com/>. Cited 30 April 2010
34. Linux Advanced routing and Traffic control HowTo. Available online: <http://lartc.org/howto/index.html>. Cited 30 April 2010
35. Lakshmi, J., Nandy, S.K.: I/O virtualization architecture for security. In: IEEE Proceedings of International Workshop on Virtualization Technology (2010)



<http://www.springer.com/978-0-85729-048-9>

Grids, Clouds and Virtualization

Cafaro, M.; Aloisio, G. (Eds.)

2011, XV, 235 p., Hardcover

ISBN: 978-0-85729-048-9