System Virtualization in the Multi-core Era - a QoS Perspective

A THESIS
SUBMITTED FOR THE DEGREE OF
Doctor of Philosophy
IN THE FACULTY OF ENGINEERING

by

J. Lakshmi

Supercomputer Education and Research Center
Indian Institute of Science
BANGALORE – 560 012

December 2010
TO

My parents for being my inspiration and guiding spirit, my husband Sreehari for his immense patience and irrevocable support, and my daughter Vasudha for her thoughtfulness and enthusiasm.
Acknowledgements

My interest in research started with a curiosity to find the behind the scene reasons for system failures, since I have been working for more than a decade in the field of systems administration. But with my job and home responsibilities, this took a back seat and kind of got buried in the daily chores. A casual remark by my then department’s Chairman, Prof. N. Balakrishnan, that I should consider research as a career option ignited this quest. For this I pen down my sincere thankfulness to him. Perhaps, without this initiation I would still be mired in the details of problem solving, although it was getting tiring and boring.

Next, the issue of how to start-off finding a good topic and a research guide who could direct me on this path was actually made easy by Prof. S. K. Nandy who spontaneously suggested several topics and related areas that could interest me. My initial hesitation was put to rest by his belief and confidence in me and there started my journey which culminated into this work. I am forever indebted to my research guide for this. This work has taught me the due diligence needed to carry out research work. He has been a tremendous guiding light during my six long years of association, specifically for the research work. He has been an exemplary person in terms of his willingness to listen and help in analyzing the issues and concerns associated with any research work. His deep insight and clarity of thought is an inspiration to learn more, in the hope of developing the same insight! He has an unlimited source of patience that helped me wade out of many big and small issues. His generosity to provide ample resources to his students has always made things easy. But for him, I do not think it would have been possible for me to carry out this work, particularly along with my other liabilities of a full-time job and a home. I would like to mention here that for any student Prof. Nandy is definitely a blessing in disguise!
A special mention here, of Prof. V. Rajaraman, is for his wonderful support and guidance. When one of my research papers got accepted to be published as a book chapter, I was really at loss on how to start. Eventually, after compilation, he very gracefully accepted to give it a read. It was indeed with great clarity he cited how the chapter could be made more readable with loosing out on details. Today, the book has been published and I am much grateful to him for that.

My research work has partly been sponsored by M/s Boeing and the technical research team has been very encouraging about this work. Periodic review meetings with them has ensured timely completion of work. Deep and inquisitive questions from the team has always forced me to evaluate this work meticulously, and also, helped me to understand the real users of technology. I am grateful to the entire Boeing team for giving me the opportunity to work with them. I would also like to thank the ANRC project team members for their unfailling support in providing all the required infrastructure for conducting innumerable experiments that made this work credible. As an extension to this team, I would like to thank all the project interns who painstakingly conducted the experiments to validate this work. I also would like to mention all those unknown reviewers whose remarks, comments, and pointers have really helped in refining the work.

Of course, my department’s previous Chairman, Prof. S. M. Rao, the current Chairman, Prof. R. Govindarajan and all my other colleagues in my department also deserve a definite mention here for their constant encouragement, co-operation and patience in allowing me to complete this work. Although, I have tried not to neglect my office work in lieu of my research urgencies, there have been several times where I could not have met the deadlines but for their support. Such support has acted as a significant motivation for me to put my heart into whatever job was at hand. It is indeed overwhelming to know that I am among such colleagues.

My lab and the room that I spent hundreds of hours pondering over unresolved issues is a place that will remain with me for the rest of my life. Many a times while doing my RTP and subsequently, the research work, I have felt that I am perhaps really aged to be with the bright and buoyant youngsters around. But they have always instigated in me the challenge to go ahead. I would like to note here my heartfelt gratitude to all the lab students, whose presence
itself was like contagious enthusiasm. Any casual or formal discussion with them has always been very refreshing and productive.

No work for sure is going to be accomplished if you do not have a happy and peaceful home. For this I must admit that I have two guardian angels in the form of my husband, Sreehari and daughter, Vasudha. I will not do justice to this work if I do not acknowledge their contribution to my progress. My husband has always been my pillar of support in any and every form; endless chauffeuring to my lab at odd hours to help me complete my experiments in time, to catering to any kind of need from cooking to engaging guests, taking care of my home responsibilities, etc., the list is endless. His commitment to my work has always been, "if you are interested, you must do it". This attitude has really helped me complete my work with ease. Next comes my little daughter. She has always been very enthusiastic about my passion towards work and has helped me in every possible way, to reduce her dependence for me, so that I can get extra hours to spend on my research. She has the thoughtfulness that is amazingly way beyond her age. But for their sustained support and encouragement I could not have completed this work.

Lastly, I inscribe here the debt I owe to my parents for being my inspiration and guiding spirit, always. I have grown up, seeing them completely dedicated to their work, never, ever compromising on their commitment to it, irrespective of how big or small the task was. They have never asked me to be something because they desire it, instead they have always emphasized that the first thing that I owe is happiness to myself. And this has taught me to do my work whole heartedly. It has instilled a quiet confidence in me that guides me to only do what is conscientiously acceptable. This principle has made my life wholesome and very happy. I would definitely wish to have them as my parents, in every life and aspire to teach my children the same things that they have taught us for years by their deeds.
Publications based on this Thesis


Abstract

Emergence of multi-core servers and the need for green computing has necessitated the resurgence of system virtualization. System virtualization has re-emerged as an answer to many critical issues being faced by the enterprise computing infrastructure. The issues that virtualization addresses are myriad, with the most popular one being server consolidation. Server consolidation deals with the problem of co-hosting multiple, independent application servers on a single physical machine. Abundance of virtualization technologies for commodity systems has given the necessary impetus for adoption of virtualization based solutions, particularly in the enterprise segment. However, most of the emerging virtualization solutions still need to address several challenges. Most prominent of the challenges, that are being faced today, are for the I/O virtualization architectures. Current technologies fall short in addressing the performance and security issues for I/O workloads on virtualized servers.

Virtualized servers allow for co-hosting multiple independent servers as virtual machines (VMs), on the same physical machine. Hence, on virtualized servers multiple VMs share some or all the physical resources. When compared to non-virtualized servers, virtualized servers show better utilization of system resources. In virtualization parlance, the number of independent servers, that can be hosted as VMs on a virtualized server, is called the consolidation ratio. The consolidation ratio depends on the machine’s capacity and the workload of the individual servers to be consolidated. Arriving at the consolidation ratio is a complex process. Majority of the capacity planning efforts have focused on the CPU component of the workload while consolidating workloads. This is effective for compute workloads. The consolidation ratio for such workloads can be arrived at by aggregating the CPU workload component of the
VMs against the system’s CPU capacity. This scenario changes when the workloads to be consolidated are composed of significant compute and I/O components. The consolidation ratio for such workloads is dictated not only by the CPU workload aggregation but also by the I/O workload aggregation against the available capacity in the system. With limited number of I/O devices on virtualized systems, particularly in multi-core servers, the consolidation ratio for I/O workloads is not only limited by the capacity but also by the performance isolation due to shared devices amongst independent VMs. In view of this, in this thesis we explore prevalent end-to-end I/O virtualization architectures with the following goals:

1. To understand virtualization effect on I/O workload performance.
2. To understand resource utilization implications for I/O workloads due to virtualization.
3. To understand effectiveness of existing Quality of Service (QoS) controls on resource usage and thereby application performance.

The key driver for virtualization adoption in data-centers, apart from software isolation, is virtual machine performance and security 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 system virtualization architectures that provide CPU workload isolation. The number of CPU-cores, in comparison to I/O interfaces, is high in the multi-core servers. This results in the sharing of I/O devices among independent virtual machines and hence, changes the I/O device sharing dynamics, when in comparison to, dedicated, non-virtualized servers. On the non-virtualized servers, all the resources, like the processors, memory, I/O interfaces for disk and network access, are architect-ed to be managed by a single OS. In such systems, solutions that optimize or maximize the application usage of the system resources are sufficient to address the performance of the application. In contrast, when independent applications are consolidated on to a multi-core 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 individual VMs, on the consolidated server while sharing a single I/O device. Prevalent I/O virtualization architectures suffer from high overheads and performance interference due to device sharing. These issues cause variability in application performance that is dependent on the nature of consolidated workloads and the number of VMs sharing the I/O device. The performance interference also causes security vulnerabilities that lead to denial of service like attacks. One way to control this variability is to impose necessary QoS controls on resource allocation and usage of shared resources. Existing technologies merely extend the system software to provide resource usage specific QoS controls. This results in coarse-grained controls that tend to be ineffective and inefficient. This behavior mandates that I/O virtualization architectures need to be re-looked at, and evaluated to ensure that the system design meets the virtualization architecture goals.

The desirable features for designing the system hardware to support virtualization architecture goals, are:

1. Efficiency: Virtual Machine should be as efficient as the non-virtualized machine, if not more!

2. Isolation: Virtual Machine is an isolated duplicate of the real machine. Hence, sharing of resources on virtualized machine should not cause any kind of interference amongst the VMs.

3. Safety: The Virtual Machine Monitor should have complete control of resources. This implies that any VM cannot access any resource not explicitly allocated to it, and it is possible for the VMM to regain control of resources already allocated to any of the VMs.

To ensure that these properties prevail, the resource management constructs used for virtualization, should be light-weight and closer to real resources. This allows for tighter resource usage controls and provides for appropriate stubs for the much desired hierarchical adaptive scheduling framework for virtualized environments. Existing hardware design is architect-ed with the goal of allowing a single OS to use, control and manage all the resources. Most operating systems are process centric and system architecture is designed to benefit this model. Prevalent general purpose operating systems project the physical or real hardware resources
using OS abstractions like processes, pages, files, sockets, packets, etc. Sharing, of the physical resource, across multiple processes or tasks occurs, within the context of the OS, using these resource abstractions. Majority of the resource control and sharing policies, are built over the OS resource abstractions and are used by the resource schedulers within the OS. System virtualization, on the other hand, by its very definition, requires every resource to support concurrency in terms of access context. This is specifically true for I/O devices since prevailing devices are not designed for concurrent access.

We re-look at the I/O virtualization architectures, with NIC as a specific example and present an end-to-end NIC virtualization architecture that aims to fulfill the above listed virtualization goals. The I/O virtualization architecture is achieved using hardware defined reconfigurable virtual device interface (HDReconfig-VDI). In this thesis, NIC is identified as the I/O device for the case-studies, to analyze the virtualization architectures, because it is a resource that has a strong potential for sharing, and higher utilization of such a device benefits many segments of users, specifically enterprise users. Also, network device sharing leads to complex usage, and ensuring performance and security isolation is a major concern here.

The design properties for the HDReconfig-VDI are:

1. Device hardware support for reconfigurable virtual device context.

2. Device time-sharing support by the device controller with variable time-slices to enable adaptive resource allocation.

3. Manifestation of virtual device over a physical device through late or dynamic-binding.

Supporting virtual device context, on a physical device, allows for native device access through the virtual device interface, and enables concurrency at the device. Concurrency on the device improves performance significantly. Native device access using the HDReconfig-VDI also improves security of the virtualized server. Time-sharing over a resource improves its utilization and also system throughput for all the VMs using the device. The biggest challenge to time-sharing is with the context switching time. If a device natively supports concurrency, context switch can be easily implemented in hardware by passing the control to the context that is ready to be scheduled. This also improves the scope for better co-ordination between
hierarchical device schedulers that are being proposed for virtualized servers and virtual machine ecosystems. Furthermore, dynamic or late-binding creates a less-cohesive virtual to physical resource mapping, that gives the flexibility to exploit unused resources and provide mechanisms for easy relocation and migration of VMs.

In the thesis, using the HDReconfig-VDI we propose an end-to-end I/O virtualization architecture. 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 virtual machine monitor (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 non-virtualized 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/hosting domain 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 order to use HDReconfig-VDI various layers involved in virtualization (the virtualization stack), like the hardware, hypervisor, and the Guest OS, need to be suitably modified. We need a standard method for evaluating the benefits of the proposed end-to-end I/O virtualization architecture. Increasing adoption of virtualization technologies bring out the need for standard methods for modeling and evaluating different technologies. Most efforts in evaluation are currently directed towards identification as well as evaluation of suitable virtualization benchmarks. This is because performance evaluation of diverse technologies is the major activity. Benchmark identification does help the user community to evaluate existing solutions. However, for evaluating architectures and changes made to the different layers of the virtualization stack, it is imperative to have a uniform framework for modeling multiple technologies to analyze their behavior. In order to compare and identify the performance bottlenecks on the virtualization stack, and how they can effect different applications it is necessary to have a testbed that helps in analyzing the various components and their associated behavior.
Modeling of virtualization environments can be complex, specifically if it involves end-to-end architectures. The basic requirement for virtualization environments is being able to capture the contention of software and device sharing, for concurrent use, which is critical for virtualization technologies. We find that layered queuing networks (LQNs) are ideal for modeling such systems without losing out in details and at the same time being able to give reasonably good estimates on performance so as to enable evaluation of end-to-end architectures. The LQNs models enable sufficiently high level of abstraction of the system under consideration which enables using them for analyzing complex systems like virtualization stacks. The LQNs used in this thesis were built using LQNs software of Carleton University. Comparing the simulations with experimental observations we have found that they can be used to model and evaluate changes and enhancement on the virtualization stack with sufficient accuracy. Hence, the proposed I/O virtualization architecture using the HDReconfig-VDI is built as a LQN model and evaluated. The evaluation is carried out by comparing the simulation results of the LQN model for proposed architecture with that of the simulation results of the LQN model of the existing Xen I/O virtualization architecture.

The evaluation for the proposed architecture is carried out systematically for the three identified metrics namely, throughput, server CPU utilization and effectiveness of resource usage specific QoS controls. On the existing technologies for I/O workloads, we observe that the application throughput on a VM is reduced and server CPU utilization is increased when moved from a non-virtualized server to a virtualized server. This is attributed to the high virtualization overheads caused by the software layers used for providing I/O virtualization. Making the hardware device virtualization aware enables the I/O device for native access by a VM, and hence these overheads are considerably reduced. The throughput improvement is about 60% for the proposed architecture, with an equivalent reduction in CPU utilization overheads due to virtualization. Also, enabling independent virtual device context on the hardware improves resource usage specific QoS controls which is reflected in the observed application throughput. Furthermore, involving the physical device in access management of the virtual device provides better security management and hence reduces vulnerabilities associated with sharing the I/O device.
By virtue of studying and evaluating I/O virtualization architectures we conclude that designing systems from an end-to-end perspective enables greater flexibility in managing resources for virtualization and delivering additional benefits of performance and security. We observe that both the characteristics, performance and security, can be handled with simple, elegant constructs that are built on hardware APIs.
List of Abbreviations and Definitions

**ABI:** Application Binary Interface.

**Active Server:** In a LQN, the active server is the one that can accept and generate requests. Hence, it refers to the middle tier application servers that can do some part processing and forward the request to other servers for subsequent processing.

**API:** Application Programming Interface.

**Application Software Dependency:** Refers to specific versions of OS and system libraries and tools required for correct functioning of the application.

**Cluster:** Group of identical servers connected through a high-speed interconnect and software to enable use as a single large computer.

**Co-hosting:** Refers to execution of different applications, each within its own VM, on a single physical machine.

**Commodity Systems:** Computer Systems manufactured by multiple vendors using components based on open standards.

**Computational Grid:** Refers to a large collection of computers linked via a global network like the world-wide web in such a way that there processing power can be harnessed for solving problems that are time-consuming or require many resources.
Consolidated server: Consolidated server refers to a virtualized server that is hosting more than one VM.

Consolidation: Refers to the co-hosting of multiple, independent application servers on a single machine using VMs.

Dedicated I/O domain: Dedicated I/O domain, alternatively called the Independent Driver Domain (IDD) or the hosting VM, is an exclusive VM that handles I/O requests to and from an I/O device using the native device driver. The VMM can also be the driver domain.

DMA: Direct Memory Access.

FC6: Fedora Core version 6 Linux Operating System.

Grid Computing: Grid Computing refers to the use of many, heterogeneous computers on a network, as a unified resource, to solve a single problem.

Grid Middle-ware: Grid Middle-ware refers to a set of co-operating programs that enable access and use of Grid Resources.

Grid Resource: An independent, allocatable unit within a Computational Grid.

Grid Scheduler: Is a software component of Grid Middle-ware that schedules Grid Applications onto available Grid Resources.

Guest OS: Guest OS is the operating system executing within a virtual machine.

HAL: Hardware Abstraction Layer.

HDReconfig-VDI: Hardware Defined Reconfigurable Virtual Device Interface.


Hypercall: Hypercall is a system call equivalent for the GuestOS to reach into the hypervisor privilege mode.
I/O: Input and Output.

IDD: Independent Driver Domain, is the device hosting domain in Xen virtualized server.

Infrastructure Cloud: Refers to provisioning of computer hardware on-demand typically as a service.

Instruction Emulation: Mechanism for imitating instructions of other device or architecture on the hosted machine. Emulated instructions are trapped and translated at runtime to that of the hosted device or architecture.

IOMMU: Input Output Memory Management Unit.


ISA: Instruction Set Architecture.

Legacy Application: An application that was written for an earlier operating system or hardware platform.

LQNs: Layered Queuing Networks refer to queuing network models that are composed of servers who are conceptually organized as hierarchical layers. Each of the layers represent a functional group. Servers belonging to a layer communicate with the servers of neighboring layers.

MSI: Message Signaled Interrupts.

MTU: Maximum Transmission Unit.

NIC: Network Interface Card.

Non-virtualized server: A non-virtualized server refers to a system that is booted using a general purpose OS like Linux or Windows.

OS: Operating System.
Para-virtualization: Para-virtualization refers to software based methods for supporting virtual devices on real devices and needs device specific modifications in the hypervisor, GuestOS and the driver domain.

Pure Server: A pure server only accepts requests and thus, is suitable for modeling hardware resources which ultimately execute the task.

QoS: Quality of Service.

Receive and Segment Offload: Receive and Segment offload refer to the functional capabilities built into the NIC card to support high-bandwidth connections. Receive offload is a technique for increasing inbound throughput of high-bandwidth network connections by reducing CPU overhead. It works by aggregating multiple incoming packets from a single stream into a larger buffer before they are passed higher up the networking stack, thus reducing the number of packets that have to be processed. Segment offload is a technique for increasing outbound throughput of high-bandwidth network connections by reducing CPU overhead. It works by queuing up large buffers and letting the NIC split them into separate packets.

Reference Server: In the context of LQNs, reference server refers to a server that generates requests only. Most clients of real life applications fall into this category.

RNV: Rendezvous communication is a type of blocked communication among servers of a queuing network. A server supporting RNV is typically composed of multiple phases of execution for a request. Blocking of the requester occurs during the phase-1 execution of the multi-phase server. Reply from server after phase-1 execution releases the requester. Subsequent to this, the server may execute the remaining phases. These later phases execute concurrently with the requester.

ROI: Return on Investment.
Single-device Single-OS usage model: This usage model is applicable to devices that are architect-ed to be accessed by a single OS. The device is transparent to, and is unaware of, concurrent access by multiple VMs.

SRNs: Stochastic Rendezvous Networks are queuing networks composed of RNV servers.

TCO: Total Cost of Ownership.

VDI: Virtual Device Interface.

Virtual Interrupts: The device interrupts used by the hypervisor to communicate to the GuestOS, for device specific action, are termed as virtual interrupts. The hardware interrupt generated by the physical device is intercepted by the VMM and after processing is forwarded to the respective VM, using a soft interrupt mechanism.

Virtualized server: Virtualized server refers to a system that is booted using a hypervisor and hosting a single VM.

VM: Virtual Machine.

VMI: Virtual Machine Interface refers to the protocol that is used by the GuestOS to communicate with the hypervisor.

VMM: Virtual Machine Monitor, or more commonly called the hypervisor, is the control program on virtualized servers that manages all system resources and replaces the generic OS of the non-virtualized server.

vNIC: Virtual NIC.
## Contents

Acknowledgements i  
Publications based on this Thesis iv  
Abstract v  

### 1 Introduction  
1.1 Re-emergence of Virtualization . . . . . . . . . . . . . . . . . . . . . 2  
1.2 Virtualization in its varied avatars . . . . . . . . . . . . . . . . . . . . 5  
  1.2.1 Processor/ISA Virtualization . . . . . . . . . . . . . . . . . . . . . . 8  
  1.2.2 HAL Virtualization . . . . . . . . . . . . . . . . . . . . . . . . . . 9  
  1.2.3 OS-Level Virtualization . . . . . . . . . . . . . . . . . . . . . . . 10  
  1.2.4 API Virtualization . . . . . . . . . . . . . . . . . . . . . . . . . . 11  
  1.2.5 ABI Virtualization . . . . . . . . . . . . . . . . . . . . . . . . . . 12  
1.3 System Virtualization . . . . . . . . . . . . . . . . . . . . . . . . . . . 12  
1.4 Performance Issues with I/O Device Virtualization . . . . . . . . . . . . 15  
1.5 Objective of the Thesis . . . . . . . . . . . . . . . . . . . . . . . . . . . 19  
1.6 Organization of the Thesis . . . . . . . . . . . . . . . . . . . . . . . . . 20  
1.7 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21  

### 2 QoS and Virtualized Servers  
2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23  
2.2 I/O Device Virtualization . . . . . . . . . . . . . . . . . . . . . . . . . . 25  
2.3 Experimental case-studies for analyzing virtualization side-effects . . . 31  
  2.3.1 Case-Study 1: Virtualization effect on Application Performance . . 38  
  2.3.2 Case-Study 2: Virtualization effect on Application Resource Usage . 43  
  2.3.3 Case-Study 3: QoS and Application Performance . . . . . . . . . . . 47  
2.4 Issues with Prevalent NIC Virtualization Architectures . . . . . . . . . . . 50  
2.5 Architecture wish-list to overcome existing lacunae . . . . . . . . . . . . 52  
2.6 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53  

### 3 Architecture for System Virtualization  
3.1 Application requirements for performance isolation on shared resources . . 55  
3.2 System Virtualization Goals and Desirable Resource Provisioning Properties . 59
CONTENTS

3.3 Review of I/O Virtualization Architectures ........................................... 66
3.4 End-to-end I/O Virtualization Architecture using Hardware Reconfigurable
    Virtual Device Interface ................................................................. 69
    3.4.1 Proposed NIC Virtualization Architecture Description ..................... 72
    3.4.2 Network Packet work-flow using the Proposed NIC Virtualization Ar-
        chitecture .................................................................................. 76
3.5 Summary ............................................................................................. 77

4 Modeling Virtual Machines using Layered Queuing Networks .................. 79
    4.1 Introduction ..................................................................................... 80
    4.2 Modeling End-to-End Virtualized Server Architecture ....................... 82
    4.3 Layered Queuing Networks using Rendezvous Communication for Modeling
        Shared Devices and Software in Virtualized Servers ............................ 83
        4.3.1 Throughput bounds for SRNs ................................................. 89
        4.3.2 Throughput measurements in SRNs ........................................ 91
        4.3.3 SRN Algorithm to calculate server throughputs .......................... 101
    4.4 Generating Layered Queuing Network Models .................................... 102
    4.5 LQN Model for End-to-end Xen Virtualized Server .............................. 104
    4.6 Summary ......................................................................................... 119

5 Evaluation of Proposed End-to-end Virtualization Architecture ............... 120
    5.1 Introduction ..................................................................................... 120
    5.2 Evaluation of Proposed Architecture .............................................. 122
        5.2.1 LQN model for the proposed architecture ............................... 122
        5.2.2 Simulation and Results .......................................................... 125
    5.3 Summary ......................................................................................... 129

6 Conclusions and Future Work .................................................................. 132
    6.1 Conclusion ..................................................................................... 132
    6.2 Lessons Learnt ................................................................................. 136
    6.3 Future Work ................................................................................... 137
    6.4 Summary ......................................................................................... 139

Bibliography ............................................................................................. 140
List of Tables

4.1 Execution service time demands for task entries, with their phases and processor deployment, used in the LQN model of Xen Virtualized Server, hosting two VMs, for *httpperf* benchmark .................................................. 109

4.2 Processor-Task assignment used in the LQN model of Xen Multi-core Virtualized Server for *httpperf* benchmark ................................................. 113
List of Figures

1.1 A Schematic Diagram of System Abstraction Layers .......................... 7
1.2 Processor/ISA Virtualization ..................................................... 8
1.3 HAL Virtualization ................................................................ 9
1.4 OS-Level Virtualization .............................................................. 11
1.5 API Virtualization ................................................................. 12
1.6 ABI Virtualization ................................................................. 13
1.7 Generic Architecture of an I/O Device in Non-Virtualized Server .......... 14
1.8 Hosted and Emulated I/O Device Virtualization Architectures .......... 17

2.1 Generic System Virtualization Architecture of prevalent commodity virtual-
ization technologies ................................................................. 26
2.2 A Conceptual Diagram of Vmware ESX Server Emulated I/O Virtualization
Architecture ............................................................................ 28
2.3 A Conceptual Diagram of Vmware Hosted Workstation I/O Virtualization
Architecture ............................................................................ 29
2.4 A Conceptual Diagram of Xen Independent Driver Domain I/O Virtualization
Architecture ............................................................................ 30
2.5 Benchmark performance comparison charts on non-virtualized, virtualized and
consolidated single-core Xen server .............................................. 40
2.6 Httpperf server throughput on Vmware-ESXi virtualized single-core server case.
The VMs are pinned to a single core while hypervisor uses all available cores 42
2.7 Httpperf server throughput on Xen and Vmware-ESXi virtualized multi-core server.
The hypervisor and the VMs are pinned to independent cores ............... 43
2.8 Benchmark resource usage comparison charts on non-virtualized, virtualized and
consolidated Xen single-core server .............................................. 44
2.9 Httpperf server %CPU utilization on Vmware-ESXi virtualized single-core server.
The VMs are pinned to a single core while hypervisor uses all available cores 45
2.10 Httpperf server % CPU Utilization on Xen and Vmware-ESXi virtualized multi-
core server. The hypervisor and the VMs are pinned to independent cores .. 46
2.11 Effect of hypervisor network bandwidth controls on application throughput for
consolidated virtualized server hosting two VMs ............................. 48

3.1 An example Application Resource Requirement tuple for a VM, expressed in
XML ....................................................................................... 57
3.2 NIC virtualization architecture supporting multiple, independent, reconfigurable
virtual-NICs ................................................................. 72
3.3 Work-flow of network I/O communication using the proposed I/O device vir-
tualization architecture .................................................. 78

4.1 Virtualization stack depicting resource abstraction and management layers .... 81
4.2 Schematic diagram of Stochastic Rendezvous Network Model showing differ-
etent interacting servers ................................................... 87
4.3 Arrival Instant Conditions at the server involved in Rendezvous Communi-
cation in a SRN ............................................................... 94
4.4 Task Interlocking due to Rendezvous Communication in a SRN ................. 96
4.5 Software contention model for a consolidated Xen virtualized server hosting
two VMs sharing a single physical NIC ............................... 103
4.6 LQN Model for consolidated Xen virtualized server supporting two indepen-
dent httpperf streams ...................................................... 105
4.7 httpperf Benchmark throughput validation charts for non-virtualized, virtual-
ized and consolidated Xen server. Comparison is between experimentally ob-
served data with LQN model simulation results ......................... 112
4.8 httpperf Benchmark CPU Utilization validation charts for non-virtualized, vir-
tualized and consolidated Xen server. Comparison is between experimentally
observed data with LQN model simulation results ......................... 114
4.9 Simulation results for network bandwidth control modeling using server think
time on Xen LQN Model .................................................... 115
4.10 Comparison graphs for httpperf achievable throughput using network band-
width controls on a Xen consolidated server with that of its simulated LQN
model ............................................................................ 117

5.1 Layered Queuing Network Model for the Proposed end-to-end NIC Virtualiza-
tion Architecture for Consolidated Virtualized Server hosting two independent
httpperf streams ............................................................... 124
5.2 Maximum throughput achievable per httpperf stream and CPU utilization for
existing Xen architecture on a multi-core server hosting two VMs each servic-
ing one httpperf client. The IDD, VM1 and VM2 are pinned to independent
cores ............................................................................. 126
5.3 Maximum achievable throughput and CPU utilization charts for a multi-core
virtualized server incorporating the proposed I/O virtualization architecture
and hosting two VMs, pinned to different cores, each servicing one httpperf
stream ............................................................................. 127
5.4 NIC QoS effect on Application Throughput for the Proposed Architecture ... 129
Chapter 1

Introduction

Emergence of multi-core servers and the need for green computing has necessitated the resurgence of system virtualization. Abundance of virtualization technologies for commodity systems has given the necessary impetus for adoption of virtualization based solutions, particularly in the enterprise segment. However, most of the emerging virtualization solutions still need to address several challenges. Most prominent of the challenges, that are being faced today, are for the I/O virtualization architectures. Current technologies fall short in addressing the performance and security issues for I/O workloads on virtualized servers. The chapter identifies these shortcomings and the need for addressing them. The chapter starts with the motivation for the re-emergence of virtualization in section 1.1. Further, the background for the thesis is set by identifying the basic characteristics of system virtualization from the other, varied virtualization mechanisms, and proceeds to analyze the prevalent system virtualization architectures, leading to the goals of the thesis. The rest of the chapter is organized as follows: In section 1.2 the context for the system virtualization architectures, as described in the thesis, is set. In this section system virtualization is defined to differentiate from the general virtualization terminology. This is followed by section 1.3, which deals with a more detailed discussion of system virtualization describing the intended goals. The section highlights I/O device virtualization architectures in the existing systems, to bring out the deficiencies. This is followed by section 1.4, which discusses the application performance issues associated with the I/O device virtualization architectures. In section 1.5 the objective of the thesis is clearly
stated and section 1.6 details the organization of the thesis. The chapter concludes with a summary of its contents, in section 1.7.

1.1 Re-emergence of Virtualization

System virtualization has, re-emerged as an answer to many critical issues being faced by the enterprise computing infrastructure. The issues that virtualization addresses are myriad, with the most popular one being server consolidation. Server consolidation is co-hosting multiple, independent application servers on a single physical machine. In today’s context, it is pertinent to seek an answer to the need for virtual machines (VMs), particularly when abundant low-cost real machines are available. Compared to expensive mainframe computers of yester years, the servers of today are definitely much faster and more affordable. As computers are becoming all pervasive in everyday life, dependence on them is increasing. Today’s corporate businesses, banks, health care, manufacturing, retail, and almost all government services are heavily dependent on software and hence, the systems hosting them. Many of these organizations go to great lengths to ensure service availability by way of dedicated, redundant servers for hosting critical services, at the cost of under utilized servers [62]. In fact, over provisioning of resources is the norm rather than an exception. Many enterprise data center surveys, on the average server utilization statistics, indicate very low (< 5%) system resource utilization [36].

While more powerful servers have become economically affordable, other factors like conflict in software dependencies, assurances for service availability, isolation of application for performance, security and fault-tolerance, etc. are the major reasons for extra resource provisioning. More often, due to the critical nature of the applications, co-hosting of multiple applications on a single server is avoided. Similarly, the availability solution for each of these applications is independent and isolated. This has led to the establishment of large data center like facilities causing server sprawl, which leads to ever increasing power, real estate and carbon footprint. The stimulus for this kind of deployment has been the decreasing cost of increasingly powerful systems. Such adoption strategies have their own disadvantages like increase in the running and maintenance costs of the compute infrastructure and associated
software, more often than not, due to the proprietary nature of the solution. Now, with green computing taking precedence, reduction of the energy, heat and real estate footprints have become the critical issues. Added to this, reduction of the total cost of ownership (TCO) is the key driving force for reuse of existing compute infrastructure. In addition, many organizations have started looking closely at the return on investment (ROI) for these data centers, specifically in the context of above described factors. The need for application and in-turn the server consolidation has always been there and the re-emergence of virtualization on commodity platforms has given the necessary drive. The wide acceptance of VMs is because virtualization provides solution to continue hosting legacy applications on a platform where innovative operating system solutions can be tried out side by side on the same system.

At the same time, a steady revolution in terms of moving away from proprietary and expensive solutions is being increasingly observed. Many organizations have already realized the benefits of using cheaper, low power, rack-mountable commodity servers as clusters. On such a cluster virtualization has come to be seen as the panacea for a majority of the data center issues [36]. The reason being, commodity clusters accommodate a large number of good performance hardware, and virtualization provides the platform for hosting any application along-with its environment. This way, legacy applications are supported seamlessly on any server, including the application’s redundancy requirements. This is perhaps an ideal setup as far as the software dependency and application performance issues are considered, since each node in the cluster could replace in-toto the old server with all its resources.

Further, with the emergence of grid computing, usability of clusters and distributed systems in solving the issues of scaling an application for larger resources and higher concurrent usage got realized. In the computational grid, each resource that is allocatable by the grid scheduler to the application, is the single physical server, also called the node, as in cluster terminology. The cluster interconnect, or the local LAN, provides the communication fabric for the grid resources, and the middle-ware is the unifying software for the distributed resources. This is akin to M:1 mapping with multiple distributed resources projected as one logical system to the application. Of course, the application has to be written to exploit such multiple resources. The basic allocation resource on a Grid is a node, which is similar to a complete system with
single or multiple CPUs having access to its own memory, disks and network interface cards. Any job or task scheduled on a node is presumed to have access to the node’s resources as if in a dedicated mode. The need for local resource sharing is restricted to the scheduled job or its tasks. Hence, most applications in such environments are programmed to optimize the node’s resources across their own tasks. Further, such architectures provide inherent security to the VMs, so long as the software hosted within the VM has been certified for required security.

This scenario is changing with the emergence of multi-core servers. Multi-core servers have a good computing power with many available processors hosted on a low power server with a small real estate footprint. These servers with system virtualization provide tangible answers to many of the enterprises compute infrastructure issues. As the number of processor increases in the servers, their capacity to co-host more applications also increase. On such servers, virtualization enables the flexibility for the application to carry its environment on any available, required subset of the system hardware. With these servers, data centers are witnessing a change in terms of projecting computing infrastructure as usable systems. Today, infrastructure clouds are emerging as multiple partitions of a physical system, aka $I:M$ mapping of resources to applications. This is because in multi-core systems many CPU cores are condensed into a node and this changes the way restricted system resources, like I/O devices, are shared. Unlike in the Grids, clouds enable co-hosting of multiple, independent applications on a physical system, possibly sharing one or more I/O devices. This sharing demands that the virtual machines support isolation without losing on performance and security.

Virtualized multi-core servers have the potential to replace multiple physical servers of a data center and hence are being adopted to consolidate independent enterprise applications on to fewer systems[42]. The multi-core servers are high in CPU resources with fewer or limited numbers of I/O devices, while a majority of the enterprise applications, have both compute and I/O components in their workloads. Consolidating such workloads on to a single physical multi-core server leads to sharing of system resources among independent applications or VMs. The success of workload consolidation depends on the extent of software and performance isolation achieved between the co-hosted applications[88].

Prevalent system virtualization architectures permit independent applications to be hosted
on a single, physical server with their complete software environment encapsulated into a virtual machine. Consolidation improves system resource utilization but induces resource virtualization overheads on the application which impacts its performance. Apart from the software isolation and improved resource utilization, the key driver 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. Sharing dynamics of I/O devices on such systems differ significantly when in comparison to dedicated servers. On dedicated servers hosting individual applications, all the resources like the processors, memory, I/O interfaces for disk and network access are architect-ed 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 on to a multi-core server, using virtual machines, performance and security interference caused due to shared resources across multiple VMs adds to the challenges. Hence, on virtualized servers, it is essential to minimize virtualization overheads and also have fine-grained resource sharing controls to support desired application performance, without losing out on server resource utilization. Almost all the prevalent system virtualization technologies have focused on preserving secure performance specific to CPU resource. On multi-core virtualized servers it is necessary to extend this to other resources, specifically the I/O devices.

1.2 Virtualization in its varied avatars

The dictionary meaning of virtual implies something that is not in fact, or real but, which is imitated or simulated. Virtualization in computing has come to signify realization of desired resource abstractions over physical resources. One example is that of virtual memory, which refers to the technique wherein an application process is made to believe that there is more memory than is physically available. Another example is Intel’s Hyper-Threading Technology
(HTT). A processor enabled with HTT is treated by the operating system as two virtual processors instead of one. Both these examples represent the virtualization of a system component or device. Extending the notion of virtualization to the complete system, a virtual machine is a duplicate of the physical machine, like that which is enabled by popular virtualization technologies like Xen [24] and VMware [93]. In such cases, virtualization is a mechanism for projection of physical resources as virtual resources. A virtual resource is defined to be a subset of the physical resource. Purportedly, virtualization allows for independent allocation of the virtual resources to build a complete system on it. This system is called the virtual machine or the VM, and each VM runs its own operating system, also called the Guest OS, and applications. Although, in both cases, virtualization relates to resource abstraction, there is a distinction in the nature of resource usability, that each of the virtualization mechanisms enables. In a broad sense, various virtualization technologies have enabled virtual resource usability either at the process level, called as process virtualization technologies or at the complete system level, called system virtualization technologies.

To understand the scope of virtualization technologies we dwell on the meaning of virtualization, in the context of resources. Any mechanism of virtualization refers to the projection of the real resources as virtual resources. At a given level of abstraction, virtualization provides an interface that maps the virtual resources defined by the interface to the underlying set of, possibly different, physical resources. As a result of which, the real system may appear as a different virtual system or even as multiple virtual systems. If we conceptualize the computing machine as a set of resources, which are projected as a stack of different abstract resource layers, as depicted in Figure 1.1, each layer manifests two distinct interfaces. The bottom interface is the mapping of this layer onto the layer below and the top interface is the projection of this layer to what is made visible in the layer above. The system hardware is composed of physical components made of the processor, or the execution hardware, memory, I/O and NIC devices connected by the system interconnect bus. This hardware supports the processor instruction set architecture (ISA) and is abstracted using the ISA layer interface. The ISA basically is the abstraction of a processor on a given hardware setup and represents specifically the processor interface. Above the ISA layer, the whole of the system is abstracted out using
the hardware abstraction language (HAL) and is represented using the HAL layer. The HAL is the abstract layer over which the basic operating system (OS) kernel sits and encapsulates the hardware specific components of the system. Both ISA and HAL represent the system in terms of its hardware components. The OS uses HAL and projects the hardware resources as abstractions that are different from the physical devices. A user application uses the system resources using the OS specified resource abstractions and interfaces. The OS provides two interfaces, called the application binary interface (ABI) and the application programming interface (API) for an application. The ABI layer provides an execution environment wherein an application built to a particular ABI can be executed over any OS supporting that ABI. The API layer is still higher than the ABI layer and provides access to using system resources through a programming environment.

A variety of virtualization technologies have emerged based on the abstract resource layer where virtualization is implemented. These different kinds of virtualization technologies can be classified as listed below:

1. Processor or Instruction Set Architecture (ISA) virtualization

2. Hardware Abstraction Layer (HAL) Virtualization
Chapter 1. Introduction

3. Operating System Level (OS-Level) Virtualization

4. Application Programming Interface (API) Virtualization

5. Application Level (ABI) Virtualization

1.2.1 Processor/ISA Virtualization

A computing machine’s hardware is built to a specific Instruction Set Architecture or the ISA. The ISA of the system is the raw hardware. Processor/ISA Virtualization refers to the idea of providing support for different ISAs on the same hardware. This allows for building or executing codes built for different ISAs on the host hardware. The main goal of doing so is for allowing cross-compilation of codes and for execution of existing codes on differing processor architectures. Recently, this technique has also been used for intrusion detection [85]. This mode of virtualization is normally achieved by a software layer, often called the emulator, that interprets instructions of one ISA to that of the host at runtime. In Figure 1.2 ISA Virtualization through a processor emulator is depicted. The dotted rectangle encloses all the system components that are virtualized using this technique. Examples of systems using ISA virtualization are Bochs[50], Qemu[34], Bird[85, 83], TransMeta’s Crusoe[29, 32], etc. Usually
Chapter 1. Introduction

Figure 1.3: HAL Virtualization

the emulator is the component of the operating system of the host machine or the virtual machine. The emulators are built using low-level hardware interfaces because of which they are not readily portable. The performance overheads of software emulators are high. TransMeta’s Crusoe processor is an exception since it adopts a hardware accelerated dynamic compiler architecture to boost performance. The emulator can be exploited for security loopholes.

1.2.2 HAL Virtualization

Hardware Abstraction Layer or HAL is a software layer between the actual hardware and the OS. HAL virtualization refers to the idea of abstracting the system architecture using an interface. Hence, HAL provides support for different hardware architectures on the same system. HAL gives the flexibility to modify the underlying system architecture without the worry of changing the OS as long as the compatibility of HAL is maintained. This way, newer designs for different components can be easily integrated into existing systems and used. Of course, there are limitations to the extent this compatibility can be maintained. The advantage HAL virtualization gives is that the OS is completely isolated from the hardware changes and with appropriate hardware support for resource virtualization, complete system can be projected as a set of virtual resources. Figure 1.3 depicts the HAL virtualization abstraction schematic. The
dotted rectangle encircles the system components involved in HAL-Virtualization and depicts the boundary above which virtualization is provided. The virtual resources are defined and managed by a layer called the Virtual Machine Monitor or the VMM which sits directly on top of the hardware. The VMM is like a very specialized operating system that provides the virtual machine interface (VMI)[18] using which multiple sets of virtual resources are provided over the real resources. Different sets of virtual resources are used to form the VM, with its own Guest OS, which can be very specific to the applications that would be launched within it. The clear advantage of this approach is that each VM is isolated from the other VMs in terms of the system and application software it hosts and thus from their failures. The disadvantage is that the VM is still susceptible to system hardware failures, which can be compensated, to some extent by the fault tolerant or fail-over solutions. Also, most of the current server architectures do not have virtualization support for all resources at the hardware level. Hence, system virtualization at HAL level is supported through software layers adding to the performance and security overheads. Most commonly used virtual machines of today belong to this group, like Vmware[93, 40], Xen[24], Denali[20, 96], Virtual Server[91], kvm [13], etc.

1.2.3 OS-Level Virtualization

Virtualization at the OS-Level is provided at the top of the host OS. In this case, both the system hardware and the OS are shared by each of the virtual machines created over the virtualization layer. The VM is implemented usually as a process in the host-OS. Figure 1.4 depicts the OS-Virtualization schematic. The dotted rectangle in the picture shows the components shared in the virtualization technique. The advantage of this mechanism is that, most OS features need not be replicated in the VMs, since the host OS mechanisms are available to all VMs. As such, the VMs and its applications are light-weight. The resource allocation and control follow the process model and can easily be developed. The issues are that this type of virtualization does not give complete isolation and is susceptible to the host-OS failures. Examples of such virtualization mechanisms are Solaris resource containers [81], Parallels Virtuozzo [89], Linux Vserver [31], etc.
1.2.4 API Virtualization

API virtualization is achieved by enabling a programming environment that is platform neutral for applications. The most popular example is Java virtual machine (JVM) [58, 87]. Figure 1.5 shows a schematic of API Virtualization. In this picture too, the dotted rectangle depicts the system components involved in virtualization mechanism. Each platform has its own JVM on which an application compiled to produce a Java byte code can be executed. The advantage that an application has is that once it is compiled, it can be run anywhere, if the JVM is present. A similar example is the Microsoft’s .net CLI [82, 90]. Parrot [9] is yet another example of API virtualization aimed at providing a platform for executing Perl programs. In any of these environments, the application executes as a process within the system and hence its isolation, from other applications, is limited to its address space and context. This mode of virtualization promotes application pervasiveness by eliminating portability issues. But, is susceptible to host system software and hardware failures.
Application level or the ABI virtualization deals with techniques aimed at providing uniformity at the binary interface. Any program developed using a development platform generates execution binaries that conform to some standard binary formats. The systems, on the other hand, support these standard binary formats. In Figure 1.6 ABI Virtualization schematic is shown. The dotted rectangle in the figure outlines all the components that are involved in providing the virtualization layer. One example of such a virtualization scheme is the Lxrun software layer that sits on top of different Unix OS and supports the Linux a.out [11] and ELF [101, 59] binary formats. This scheme is also vulnerable to the host system software and hardware failures.

1.3 System Virtualization

In this thesis, virtualization refers to complete system virtualization, as defined by HAL-Virtualization. In such virtualization schemes the physical resources are under complete control of the VMM. The VMM projects the physical resources as a set of virtual resources that form the basis for the logical system over which the VM is built. In[69] Goldberg and Popek
define a virtual machine as follows: "A virtual machine is taken to be an efficient, isolated duplicate of the real machine." This definition is valid in the context of a VMM. The VMM is introduced as the software that has complete control of the system resources and provides an environment for the execution of programs in the VM that is identical to the original machine (i.e., without the VMM). This definition implies that the following holds true for a VM built over a VMM on a physical machine:

1. Efficiency: All non-privileged instructions of a program executing from within a VM execute directly on the hardware without any intervention of the VMM.

2. Resource Control: The VM can only use the resources allocated to it, it cannot change the resource allocations made to it. For any resource allocation changes, it has to seek VMM intervention.

3. Equivalence: A program executing within a VM performs in a manner that is indistinguishable, except for timing and resource availability problems, from that of when it is executing directly on the system without the VMM.

By virtue of this definition, a VM provides a private, secure and reliable computing environment and cannot be compromised by the operation of another VM. Popek and Goldberg
go on further and prove that for any VMM controlling the hardware and satisfying the above properties for the VMs controlled by it, allows for the construction of a virtualized system. The key factors that were considered in proving this theorem were ignoring of I/O and peripheral devices and their usage by the VMs. Only the CPU and memory were considered as part of the system resources and the access to these resources by the VMs was taken care of. Although in the paper the authors did not state the impossibility of extending the notion of virtualization by including the I/O devices, they did so for the sake of simplicity. However, this theory can be easily extended to the I/O devices. We illustrate the same using the example of memory mapped I/O devices. Figure 1.7 depicts the generic I/O device architecture in a non-virtualized server. Each I/O device has a device controller that has local memory that is used to buffer data transfers from and to the device. In modern systems this device memory can be mapped to the OS kernel address space and is called the I/O Device’s mapped memory. Any data movement to or from the device happens by using direct memory access (DMA) between the kernel’s I/O buffers and the I/O device’s mapped memory.

In many operating systems, access to I/O devices is routed through the OS kernel since all instructions used for I/O operations on the devices are privileged. In the case of virtualized servers, this makes the I/O devices to be intuitively accessed through and managed by the
VMM. Hence, logically any VM using an I/O device now has to redirect all its I/O calls through the VMM. Once the device data is transferred to the kernel’s I/O buffers, this data has to reach the requesting process, which in the case of virtualized servers, is hosted within a VM. This means that any data read or write operation will necessitate the data page to be either copied or shared between the VM and VMM. This is a performance degradation operation since the VM and VMM do not share the same address space. In order to ensure data integrity and isolation, the page transfer activity is a co-ordination between the VM and VMM and is a costly operation if achieved purely though software. Furthermore, the serialization that occurs, due to data transfer routed through the VMM, leads to loss of usable device bandwidth by the VM. This indirect way of I/O device virtualization makes it difficult to implement quality of service (QoS) controls specific to resource usage. On such systems the QoS controls get specified in resource abstraction layers much above the physical device. Hence, granularity of resource controls is very coarse, particularly when viewed from the perspective of resource provisioning on shared devices. Further, since the device access path is shared along-with the device, the security vulnerabilities associated with the device software affects all the VMs sharing the device.

1.4 Performance Issues with I/O Device Virtualization

The key driver for virtualization adoption in data-centers will be security and performance isolation of VMs on a consolidated server. This is essential, particularly for enterprise application workloads, like database, mail, and web-based applications, which have a significant amount of both CPU and I/O workload components [42]. Current commodity multi-core technologies have system virtualization architectures that provide CPU workload isolation [30, 52, 51, 94]. However, in multi-core servers the number of CPU-cores is high in comparison to the number of I/O interfaces. This necessitates 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 architect-ed to be managed by a single OS. On such systems, so-
lutions that optimize or maximize the application usage of the system resources are sufficient
to address the performance of the application. When several, independent applications are con-
solidated on to a multi-core server, using virtual machines, performance interference caused
due to shared resources across multiple VMs adds to the performance and security challenges.
The challenge is in ensuring secure performance of the independent I/O intensive applications,
hosted inside isolated VMs on the consolidated server while sharing a single I/O device.

In prevalent virtualization technologies, CPU virtualization is achieved using processor
scheduling techniques. The context switch latencies are further reduced by using hyper-
threaded and multi-core processors with support for VM specific processor affinity. Further-
more, processor cache related latencies due to virtualization are being tackled using VM spe-
cific data placement algorithms [52]. With regard to memory virtualization, existing virtual-
memory techniques are robust and can be easily extended to support virtualization. Early-on,
virtualization technologies supported static physical memory partitions to VMs but the re-
cent trend is towards sharing identical memory pages [92, 39, 67], compressing infrequently
used pages [39] and dynamic sizing of memory partitions using some clever techniques of de-
allocation of unused memory from idle VMs [102]. I/O device virtualization has a different
history. I/O devices in traditional operating systems have always been under exclusive control
of the OS kernel. This was done to ensure data protection and integrity. Many of the current
virtualization technologies are extension of traditional operating systems to support concurrent
access to devices. Since the hardware was built to support single-OS access, the virtualization
features are implemented in software. I/O device virtualization did not really receive the at-
tention it deserves. I/O device virtualization in prevalent technologies is either supported by
device emulation by the VMM or by hosted device using para-virtualization [62]. Figure 1.8
clearly depicts basic differences between these two techniques.

Para-virtualization refers to the device virtualization mechanism wherein actual device ac-
cess is managed by a privileged software like the VMM or a VM. The Guest OS and the
privileged software co-ordinate data access and device management to achieve optimal per-
formance. This approach incurs modification of the Guest OS code based on the interaction
needed for the device access. A typical example is the para-virtualized NIC in the Xen hypervisor. In Xen, the NIC is allowed device access from the Independent Driver Domain (IDD), which is the device hosting VM, using the native or the physical device driver. The data is then transferred from or to the requesting VM using software methods through para-virtual drivers. The para-virtual driver has two end-points, one in the IDD and the other in the Guest OS of the device using VM. In Xen the two end-points are connected through event channel communication mechanism. Para-virtualization typically involves support for device virtualization using page transfers through address translation mechanisms. The data is copied from the device memory to the IDD’s I/O buffer. The I/O buffer is then shared with the VM for which the data is destined, instead of copying. Thus actual data transfer is done only once, and hence this method tends to be efficient when compared to emulation. The problem with para-virtualization is that it involves Guest OS code modification, and inhibits support of legacy environments.

Emulation achieves virtualization by using instruction translation. Desired hardware in the VM is emulated over the underlying physical hardware with the help of VMM. Emulation is the earliest method for virtualization and is popular since it can support unmodified Guest OS codes. The consequence of emulation is that performance overheads are quite high because of instruction translation.
In either case, virtualization implementation using software methods imposes performance overheads, and this is evident in the current technologies. Prevalent virtualization architectures suffer from two 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.

Both the 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. Current I/O device virtualization architectures do not support resource usage specific QoS controls at the device. This support is provided much higher in the device and resource abstraction layers of the OS. This is because of the evolution of current virtualization architectures from the single-hardware single-OS system model. As a result, the QoS controls are insufficient and lead to loss of usable device bandwidth. The loss is highly dependent on the nature of workloads sharing the device, which makes device sharing problem non-deterministic. Ideally, the QoS controls should ensure that:

- There is no loss of application performance when hosted on virtualized servers because of shared resources.

- Any spare resource is made available to other contending workloads.

In the thesis, we explore the I/O device virtualization architectures in the context of safe performance isolation and desired QoS controls for VMs sharing common devices and propose and evaluate an end-to-end virtualization architecture that addresses the lacunae observed in prevalent technologies.
1.5 Objective of the Thesis

This thesis identifies the issues of I/O resource sharing on application performance in virtualized servers and provides a solution by proposing an end-to-end I/O device virtualization architecture. The solution is arrived at by the following steps:

1. Representative benchmarks for enterprise workloads are identified. The selection is based on the fact that enterprise applications are built using client-server architecture with a web-interface and are composed of both compute and I/O workloads. Systematic application characterization is performed on prevalent virtualization technologies, from a resource usage perspective, to support application specific throughputs. The characterization is essential to understand the resource requirement vis-a-vis application’s performance and the associated change when moved from a non-virtualized to a virtualized server.

2. Benchmark characterization identifies that, for I/O workloads, virtualization overheads manifest as increased CPU utilization to support equivalent throughputs as in comparison to non-virtualized server. Investigation and analysis of prevalent I/O virtualization architectures reveal that sharing of I/O devices involves sharing of I/O access paths also. Hence, serialization occurs at the device and on its access path. As a consequence, effective usable device bandwidth on the virtualized server is reduced. This impacts the maximum achievable application performance on the virtualized server. The thesis identifies that I/O device sharing is associated with the device access path sharing also. This is because the current I/O virtualization technologies build over ”single-OS for single hardware” model. To eliminate this overhead, the I/O devices should be virtualization aware so that the virtualization architectures can provide direct device access to the virtual machines sharing the device.

3. The thesis also identifies that device sharing leads to performance and security interference among all the VMs sharing a common device. To support guaranteed application performance on virtualized servers, resource allocation mechanisms based on application’s desired QoS, are needed. Existing virtualization technologies support resource
allocation QoS mechanisms in software at a much higher abstraction layer than the physical device. This leads to unpredictable performance that is a function of the workloads sharing the I/O device. The closer these mechanisms are to the resource, flexible is the support for sharing. Ideally, the QoS mechanisms should guarantee specified resource to the application, with the unused resource being made available to other contending or demanding VMs. This yields performance guarantees without losing out on resource utilization. The thesis proposes designing future I/O devices to provision for resource allocation specific QoS hooks on the device itself, and re-architect end-to-end virtualization architectures to exploit them in guaranteeing application performance.

4. The thesis presents an end-to-end I/O virtualization architecture that uses virtualization aware I/O devices with device level resource provisioning QoS controls. The architecture thrust is on providing application performance isolation along with improved resource utilization on a consolidated multi-core virtualized server. This is achieved by using reconfigurable device partitions as virtual devices with native access to these virtual devices from within the VMs. The virtual device specifications can be arrived at based on the QoS requirements of the VM.

1.6 Organization of the Thesis

This thesis attempts to identify the issues in prevalent I/O device virtualization architectures with the goal of providing safe performance isolation to the device sharing VMs. The organization of the thesis is as follows:

Chapter 2 presents an overview of prevalent I/O device virtualization architectures with specific benchmark case-studies to understand its behavior on the virtualized server. Case studies are conducted to observe the benchmark performance on virtualized server when compared to non-virtualized server, to explore the performance interference among two VMs sharing common I/O device and to evaluate the effectiveness of existing QoS controls in guaranteeing desired application performance. These case studies help in identifying the issues in existing I/O device virtualization architectures.
Chapter 3 proposes an end-to-end I/O device virtualization architecture to overcome the limitations described in chapter 2. The aim of this architecture is to minimize performance interference among VMs sharing common I/O devices without losing on the total I/O device utilization. To achieve this the architecture justifies the need for native device access to the VMs with fine-grained resource provisioning QoS controls. Further, in the chapter end-to-end NIC virtualization architecture is described as a representative example of the proposed architecture. It details the necessary changes required to the NIC hardware, the VMM and the VM to support the proposal. End-to-end packet workflow is illustrated to highlight how the architecture handles data reception and transmission through its various components.

Chapter 4 discusses the need for required tools to evaluate end-to-end virtualization architectures and justifies the choice of Layered Queuing Networks (LQNs) to model them. A description of LQNs and software used for LQN generation is presented. This is followed by enumerating the steps for generating LQN model for existing Xen virtualization architecture. Validation of the LQN model against observed results is presented to highlight the pros and cons of using LQNs for architecture evaluation.

Chapter 5 describes the LQN model for the proposed architecture and presents the results obtained for the three case-studies used to evaluate existing architectures. The evaluation clearly highlights the benefits of the proposed architecture.

Chapter 6 concludes on the architecture proposal and observed results highlighting the improvements. Further, the chapter explores the future directions for the undertaken study, in the context of the applicability to the current technologies and their burning issues.

1.7 Summary

The chapter introduces the need for virtualization as an answer to the existing data center issues. Further, it dwells on the concept of virtualization, in its various manifestations and focuses on system virtualization. System virtualization description is extended to I/O devices to highlight the possible techniques in current systems and also the ensuing drawbacks. Further,
the direct effect of the I/O virtualization architectures on application performance is evaluated. Based on this, the objectives of the thesis are clearly highlighted. The chapter ends with discussion of the organization of the thesis and how it achieves its objectives.
Chapter 2

QoS and Virtualized Servers

Emerging trends in multi-core servers, with system virtualization as the enabling technology, promises to be the panacea for all data-center issues. 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 secure performance isolation of VMs on such servers. The consolidation focus is on enterprise applications that are a mix of compute and I/O workloads. An analysis of prevalent, popular system virtualization technologies is presented in this chapter, with a view towards identifying issues for providing safe application performance isolation.

2.1 Introduction

Virtualized servers allow co-hosting of multiple independent servers on the same machine, because of this, on virtualized servers, different VMs share some, or all, of the physical resources. When compared to non-virtualized servers, virtualized servers show better resource utilization [42]. In virtualization parlance, the number of independent servers, that can be hosted as VMs, on a single virtualized server, is called the consolidation ratio. The consolidation ratio depends on the machine’s capacity and the workload of the individual servers to be consolidated. Arriving at the consolidation ratio is a complex process and involves the task of apportioning resources to the target workloads. This requires a detailed exercise of
workload characterization to arrive at the resource requirements with regard to CPU, memory bandwidth, I/O bandwidth, disk space, etc. Majority of the server consolidation efforts focus on the CPU component of the workload [25]. But, this is effective only for compute intensive workloads. For such workloads, consolidation can be achieved at by aggregating the CPU utilization component of the VMs, against the system's CPU capacity. The scenario changes, when the workloads to be consolidated are composed of significant compute and I/O components. The consolidation, for such workloads, is dictated not only by the CPU workload aggregation, but also by the I/O workload aggregation, against the available capacity of the system. With limited number of I/O devices on virtualized systems, particularly in multi-core servers, the consolidation of I/O workloads is limited by the capacity of the I/O device and the performance isolation offered, while sharing it among multiple independent VMs. In view of this, in this chapter, we explore prevalent end-to-end I/O virtualization architectures, with the following goals:

1. To understand virtualization effect on I/O workload performance.

2. To understand resource utilization implications for I/O workloads due to virtualization.

3. To understand effectiveness of existing Quality of Service (QoS) controls on resource usage and thereby application performance and security.

The chapter is organized as follows. Section 2.2 describes the various reported techniques in literature for I/O device virtualization. Section 2.3 details the evaluation of, predominantly used, I/O device virtualization architectures for representative enterprise workload benchmarks. The evaluation is carried out to understand the three exploration goals listed above. The chapter concludes with a clear listing of the lacunae observed in the prevalent I/O device virtualization architectures, followed by a wish-list of what is desirable.
2.2 I/O Device Virtualization

Virtualization technologies encompass a variety of mechanisms to decouple the system architecture and the user-perceived behavior of hardware and software resources. A generic architecture of system virtualization, implemented in prevalent systems, is given in Figure 2.1. 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 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, as compared to the CPU.

Among the prevalent popular technologies, there are two basic modes of I/O virtualization, namely, full system virtualization as in Vmware [41] and para-virtualization as in Xen [24] and Denali [96]. In either case, system virtualization is enabled by the virtual machine monitor (VMM), also known as the hypervisor, that provides the resource management functionality across multiple VMs. In full system virtualization complete hardware is replicated, virtually. Instruction emulation [61] is used to support multiple architectures on the physical hardware. In emulated mode of access, each VM sharing the physical device has a device driver that is implemented using emulation over the physical device driver hosted in the I/O domain or the VMM. The emulated driver adheres to a generic, commonly used device driver and thus, allows for unmodified Guest OS in the VM. Because emulation involves instruction translation, it incurs high performance overheads. The advantage is that it enables unmodified guest operating systems (Guest OS) to execute inside the VM.

In para-virtualization the Guest OS is also modified suitably to run concurrently with other VMs on the same hardware. Para-virtualized mode of access is achieved using a virtual device driver along with the physical device driver. The physical device driver is hosted inside the VMM or a hosting VM, also called the dedicated I/O domain. This 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. The virtual device driver forms the para-virtualized interface to the
physical device inside a VM and requires modification of the Guest OS. The para-virtualized interface depends on the technique used to create virtual device over the physical device and allows for good optimization of device and data access. Hence, para-virtualized drivers offer good performance. However, both these modes provide data protection and integrity to independent VMs, but suffer from loss of performance and usable device bandwidth.

Historically, 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 [77]. A dedicated software entity, called the I/O domain is used to perform physical device access and management. The I/O domain can be part of the VMM or be an independent domain.

Example of the VMM hosting the I/O domain is the Vmware-ESX server [17, 16]. The VMM has full control of the hardware device and hosts the physical device driver of the I/O device. The Guest OS of the VMs host emulation device driver of supported generic and commonly used I/O devices of the same class. Every time a VM seeks access to the I/O device, the I/O requests trap through the emulated device driver to the VMM, which then performs the I/O activity on behalf of the VM. This involves context switches between the VMM and VM.
for each I/O instruction and hence, results in high performance and resource overheads. Figure 2.2 depicts the conceptual diagram of the *Vmware* emulated device architecture supporting device emulation through the *ESX* hypervisor.

I/O devices, such as network interface controllers (NICs) and disks, are presented as virtual devices to VMs running on the *ESX* Server. A standard virtual device is exported to the Guest OS, regardless of the underlying physical hardware. For storage controllers, *ESX* Server emulates *LSI Logic* or *Bus Logic SCSI* devices, so the corresponding driver loaded into the Guest OS will be either an *LSI Logic* or *Bus Logic* driver [16].

The execution workflow of an I/O request, issued by the Guest OS in the VM, is as follows:

- An I/O request, that is issued by the Guest OS, first reaches the driver in the virtual machine.

- The driver typically turns the I/O requests into accesses to I/O ports to communicate to the virtual devices using privileged *x86/x86-64 IN* and *OUT* instructions.

- The privileged instructions are trapped by the VMM, and then handled by device emulation code in the VMM, based on the specific I/O port being accessed.

- The VMM then calls device independent network or disk code to process the I/O. For disk I/O, *ESX* Server maintains a queue of pending requests, per VM, for each target *SCSI* device. The I/O requests are then sent down to the device driver loaded into *ESX* Server for the specific physical device.

Examples of independent I/O domain are the independent driver domain (IDD) of *Xen*[24] and *Vmware’s*[41] hosted workstation. In either case, the I/O devices are private to the I/O domain and memory accesses by the devices are restricted to that domain. Any VM seeking access to the device has to route its request through the I/O domain. Hence, the request has to pass through the context switch and address translation barriers of the IDD and VM. *Xen-IDD* uses para-virtualization technique for virtualizing I/O devices, whereas *Vmware* workstation uses emulation for I/O virtualization.
Figure 2.2: A Conceptual Diagram of *Vmware ESX* Server Emulated I/O Device Architecture (source credit [17])

Figure 2.3 depicts the hosted I/O domain architecture of the *Vmware* workstation. *Vmware* workstation has a hosted architecture that allows it to co-exist with a preexisting host operating system, and rely upon that operating system for device support. This architecture allows *Vmware* to cope with the diversity of hardware and to be compatible with preexisting or legacy software. *Vmware* workstation installs like a normal application on an operating system, known as the host operating system. When run, the application portion, called the *VMAApp*, uses a driver loaded into the host operating system, called the *VMDriver*, to establish communication with the privileged VMM component, the device driver, that runs directly on the hardware. From then on, a given physical processor is executing either the host world or the VMM world, with the *VMDriver* facilitating the transfer of control between the two worlds. A context switch between the VMM world and the host world involves saving and restoring all users and system visible state on the CPU, and is thus more heavyweight than a normal process switch. Whenever the Guest OS performs an I/O operation, the VMM will intercept it and switch to the host world rather than accessing the native hardware directly. Once in the host world, the *VMAApp* will perform the I/O on behalf of the virtual machine through appropriate system calls.
For example, an attempt by the guest to fetch sectors from its disk will become a `read()` issued to the host for the corresponding data. The VMM also yields control to the host OS upon receiving a hardware interrupt. The hardware interrupt is reasserted in the host world so that the host OS will process the interrupt as if it came directly from hardware.

![Diagram of Vmware Hosted Workstation I/O Virtualization Architecture](source_credit[41])

Figure 2.3: A Conceptual Diagram of Vmware Hosted Workstation I/O Virtualization Architecture (source credit [41])

Another example of hosted I/O domain is the IDD of Xen. It is different from Vmware hosted workstation in the sense that it adopts para-virtualization for virtualizing I/O devices. Figure 2.4 illustrates Xen-IDD architecture. Xen modifies the Guest OS to be virtualization-aware and presents it with a slightly modified x86 architecture, provided through the hypercall API. This provides equivalent, although not identical, functionality with explicit calls into the hypervisor. The Guest OS changes are limited to its architecture-dependent modules. As an example, for Linux, the Xen hypercall API takes the form of a jump table populated at kernel load time. When the kernel is running in a native implementation (i.e., not atop a para-virtualizing hypervisor), the jump table is populated with default native operations; when the kernel is running on Xen, the jump table is populated with the Xen hypercalls. This enables the same kernel to run in both native and virtualized forms, with the performance benefits of para-virtualization but without the need to re-certify applications against the kernel.

I/O virtualization in a para-virtualizing VMM such as Xen is achieved via a single set of
drivers. The *Xen* hypervisor exposes a set of clean and simple device abstractions, and a set of drivers, for all the hardware on the physical platform, and is implemented in a special domain, called the Independent Driver domain (*IDD*), outside the core hypervisor. These drivers are offered via the hypervisors abstracted I/O interface for use within other VMs, and thus, are used by all Guest OSes. In each *Xen* Guest OS, simple para-virtualizing device drivers replace hardware-specific drivers for the physical platform. Para-virtualizing drivers are independent of all physical hardware but represent an individual type of device (e.g., block I/O, ethernet, etc.). These drivers enable high-performance, virtualization-safe I/O to be accomplished by transferring control of the I/O to the hypervisor, with no additional complexity in the Guest OS. *Xen* uses a generalized, shared-memory, ring-based I/O communication primitive to connect between the para-virtualized and native device driver, and is able to achieve high throughputs by batching requests.

![Diagram of Xen Independent Driver Domain I/O Virtualization Architecture](source_credit[28])

To understand the benefits or adverse effects of the emulated and para-virtualized I/O devices, several experimental case studies are presented, for a chosen set of enterprise workload benchmarks. Details of the benchmarks and the experiments are the subject of discussion in the following sections.
2.3 Experimental case-studies for analyzing virtualization side-effects

In current virtualization technologies, one extra level of indirection is used to facilitate multiple VMs on a single physical machine. This translates into performance and resource overheads, specifically for all privileged instructions like the I/O device calls. It is worthwhile to understand how this affects system and as a result the application behavior on a virtualized server.

The focus of the case-studies here are enterprise workloads that are composed of both compute and I/O workload components. Enterprise workloads are typically composed of web based application interfaces connecting to back end application servers who host mail, database or web servers. Often, the queries are in the form of http requests that trigger a reply from the application server. The reply generation might involve computation and data fetching from storage. The reply is communicated back to the user via the server network interface and communication fabric connecting the client and the server.

It has been reported that CPU workloads perform near to native performance when moved to virtualized servers [30, 51, 52, 94]. This is because a majority of the CPU instructions execute directly on the hardware when issued from within the VM. Only specific instructions, that which are involved in state change or control related, are virtualized. Hence, it is expected that the CPU workload behavior will not exhibit a major change on virtualized servers. This has been demonstrated on multiple platforms like Denali [96], Xen [24], Vmware [40], etc.

Coming to I/O workloads, the behavior is expected to see a difference, since almost all I/O instructions are privileged instructions. In non-virtualized servers all I/O instructions are executed by the privileged OS kernel on behalf of the requesting processes. This is due to the simplistic I/O device model wherein the device is designed to be controlled and accessed by a single OS. The notion of multiple processes accessing the device is transparent to it and this abstraction is supported by the OS kernel hosting it. The I/O device sharing models, being supported by the prevalent virtualization technologies, have evolved from this ”single-device single-OS” model. Hence, the virtualization layer builds the abstraction of virtual devices and manages the multiplexing and de-multiplexing of the device traffic of the multiple VMs
sharing it. The physical device is still unaware of the concurrency of device usage. Even in a non-virtualized system the latency involved in I/O device access overlaps the computation involved in the multiplexing and de-multiplexing of data to different processes. Hence, existing virtualization technologies support I/O device sharing across multiple VMs using software mechanisms. Because of this approach, serialization occurs not only at the device but also within the software layers used to access the device. Further, the shared device’s access software makes all the VMs, that are sharing the device, vulnerable to failures associated with it.

In virtualized servers, disk devices are shared differently compared to NICs. In the case of disk devices, a disk partition is exposed as a file-system that 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 movement to and from the filesystem is synchronized using the file-system buffer cache that is resident within the VMs 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. And 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 file-system policies get to be implemented within the software layers of the VMM or the hosting VM. 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 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 results in 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, which degrades application performance. It is expected that the network interface card (NIC) sharing will be more critical to application performance when compared to storage device sharing. This is because in the case of NIC sharing, single device would be handling intermixed packet streams of different VMs and the existing sharing strategies would affect the stream at packet level. Hence, in this thesis we concentrate on analyzing and addressing the NIC sharing dynamics on virtualized servers to highlight issues associated with I/O device virtualization.

In the following subsections, we discuss experimental studies conducted to address the goals listed in the beginning of the chapter using two popular virtualization technologies, namely *Xen* and *Vmware-ESX* virtualized servers. The *Xen* virtualization solution supports para-virtualization of NIC, whereas, *Vmware-ESX* solution supports NIC virtualization using device emulation. Hence, by choosing to evaluate these two technologies, we get to understand the pros and cons of the prevalent I/O device virtualization techniques.

All benchmark servers, execute on a *Dell* system with either *Intel Xeon* 1.8GHz processor or *Intel Core 2 Duo* 1.8GHz processor. On the *Xeon* server hyper-threading feature is disabled for the experiments and hence only one physical processor is available to the VMM. Most single-core experiments are conducted on this platform. The *Core 2 Duo* system has two physical cores on the same CPU and the multi-core experiments are conducted on this server. Other system resources for both the machines are same, like 2GB RAM, 10/100/1000 Mbps NIC and a local disk of 140GB divided into individual disk partitions of 35GB each. Independent disk partitions are used to host individual VMs. The NIC is configured to 100 Mbps full-duplex connection, to enable benchmark loads in the range that saturate the server without triggering *SYN-FLOOD* errors. The *Xen* 3.0.3 version hypervisor is used and is hosted in *Fedora Core 6 (FC6)* Linux OS. All the VMs are booted into FC6 *Linux*. The *Vmware-ESXi* version 3.5 hypervisor is used to host VMs for the *Vmware* specific experiments. For NIC virtualization on the *Xen-IDD* server, bridging is used to support the virtual NICs. *Xen* 3.0 version is the basic version supporting NIC virtualization using the IDD. Subsequent *Xen* releases have incorporated software based optimizations in the path of network packet delivery to the VM, which improve the performance to some extent but the basic architecture of NIC
virtualization and its associated problems remain [97]. Hence, although the experimental data is on the slightly dated version of Xen, it proves to demonstrate the architectural issues. In the case of VMware-ESXi server, the hypervisor version is the latest available at the time of reporting. ESXi uses a software switch to emulate virtual NICs over the physical NIC. The physical NIC is hosted by the hypervisor.

In all the case-studies, the experiments are conducted to understand the effect of resource virtualization on application performance and its resource usage. In each case, the behavior of the benchmark is analyzed under three environments, namely non-virtualized, virtualized and consolidated server. The non-virtualized server is the standard Linux server hosting the benchmark server component. The benchmark results obtained in this case serve as the baseline for the benchmark performance on the server. In the case of virtualized server, the system is booted into a hypervisor of the desired virtualization technology (Xen or VMware-ESXi) and a single VM is created, which hosts the exact replica of the non-virtualized Linux server. Here the benchmark server component is hosted within the VM, and the benchmark results indicate the effect of virtualization of the server. The third environment is that of consolidated server wherein two VMs, created identical to the VM in the virtualized server, share a single NIC. In this case, server component of the benchmark is hosted on each of the VMs, and independent clients make requests to each of the servers. The benchmark results, in this environment, allows us to study the effect of NIC sharing on application behavior. For each of the benchmarks, the comparison is between a non-virtualized server with that of a virtualized and the consolidated server. In some charts an additional graph called "Total-Consolidated" is also shown. This is the aggregation of the metric to include performance of both the VMs put together on the consolidated server. This graph is used as additional information to illustrate the consolidated server’s capability with regard to the aggregated workload from both the VMs.

As a representation of enterprise workloads we choose three benchmarks, namely, netperf, httperf and surge. All three benchmarks exercise the NIC, and the associated network stack. While running the benchmarks, the focus of the benchmark runs is to measure performance for a request/response sequence. A request/response sequence is taken as the analogy to the user interaction with the application server in an enterprise environment. The performance metric
of interest is the request/response throughput that the server can support.

**Netperf**

*Netperf* [45] is a benchmark that can be used to measure various aspects of networking performance. Its focus is on bulk data transfer and request/response performance using either *TCP* or *UDP* and the *Berkeley Sockets* interface and *Unix Sockets* domain. *Netperf* is designed around a basic client-server model. There are two executables namely, *netperf* and *netserver*. *Netperf* is the client side program and communicates with the *netserver* program that resides on the server, being invoked by the remote system’s *inetd* or equivalent. When *netperf* executes, it establishes a control connection to the remote system. This connection will be used to pass test configuration information and results, to and from the remote system. Regardless of the type of test to be run, the control connection will be a *TCP* connection using *BSD* sockets. The control connection can use either *IPv4* or *IPv6*. Once the control connection is up, and the configuration information has been passed, a separate *data* connection will be opened for the measurement itself using the API’s and protocols appropriate for the specified test. When the test is completed, the data connection will be torn-down and results from the *netserver* will be passed-back via the control connection and combined with *netperf*’s result for display to the user. *Netperf* places no traffic on the control connection while a test is in progress.

In the benchmark studies that follow, we report the *netperf Connect-Request-Response* for the *TCP* protocol (*TCP_CRR*) test results. In this test, for a given request and response size, maximum achievable throughput is reported as transactions per second. Throughput is measured as the number of responses the server can generate, without any loss, for a given request and response size. The test is synchronous, with one transaction at a time, and is particularly sensitive to the path-length of the network stack. The test reports its results as transactions per second where a transaction is defined as the complete exchange of establishing a connection, exchanging a single request/response transaction, and tearing-down that connection. A *TCP_CRR* test is very much like what happens in an *HTTP* 1.0 or *HTTP* 1.1 connection when *HTTP Keepalives* are not used. Since in this test, the request and response sizes can have an effect on the performance, the benchmark is executed for different sizes.
Httperf

Httperf [65, 27] is a tool to measure web server performance. It speaks the HTTP protocol both in its HTTP/1.0 and HTTP/1.1 flavors and offers a variety of workload generators. While running, it keeps track of a number of performance metrics that are summarized in the form of statistics that are printed at the end of a test run.

The most basic operation of httperf is to generate a fixed number of HTTP GET requests and to measure how many replies (responses) came back from the server, and at what rate the responses arrived. This rate of responses or replies per second is measured as the throughput of the server. To obtain correct results, it is necessary to run at most one httperf process per client machine. Also, there should be as few background processes as possible both on the client and server machines.

Since the machine that httperf runs on has a finite set of resources available, it can not sustain arbitrarily high HTTP loads. One limiting factor is the total number of TCP port numbers available at a time. Since on most UNIX-like systems it takes one minute for a TCP connection to be fully closed (excluding the TIME_WAIT state), the maximum rate a client can sustain is at most 1,000 requests per second. The sustainable rate is often lower than that. Much before httperf can saturate the TCP ports, it is likely to run out of file descriptors (one file descriptor is used for each open TCP connection). Before configuring httperf, it maybe necessary to set the kernel parameter for the number of open file descriptors to a sufficiently large number so that httperf does not report fdunavail errors. For all the machines, on which the experiments are carried out, this number is set to 165535.

One more problem that httperf is often plagued with, is that of too many open files. This is due to the poor response from the server. To handle this httperf provides the option of -timeout, which is used to set a timeout for all communications with the server. If the server does not respond before the timeout expires, the client considers the corresponding session, connection, or call to be “dead”, closes the associated TCP connection, and increases the clienttimeout error count. Timeouts allow httperf to sustain high offered loads even when the server is overloaded. Timeouts do not guarantee that a client can sustain a particular offered load. There are many other potential resource bottlenecks. For example, in some cases the
client machine may simply run out of CPU time. To ensure that a given test really measured
the server’s capabilities and not the client’s, it is a good idea to vary the type and number of
machines participating in a test. If the observed performance remains the same when the types
and number of client machines is varied, the test results are likely to be valid. For all the \textit{httpperf}
experiments described in this thesis, the timeout value used is 5 seconds. To eliminate effect
of client machine limitations, three different client machines are used as \textit{httpperf} clients.

\textbf{Surge}

\textit{Surge} \cite{23} is a tool that is an acronym for Scalable URL Reference Generator and gener-
ates references matching empirical measurements of web workloads based on the following
parameters:

1. server file size distribution,

2. request size distribution,

3. relative file popularity,

4. embedded file references,

5. temporal locality of reference, and

6. idle periods of individual users.

The workload generated by \textit{surge} is representative of web workload and includes the pauses
and bursts of typical web traffic. \textit{Surge} mimics the actual web workload using an analytic
approach that builds by combining the studied behavior of the above listed parameters. To
generate \textit{surge} specific workload, a database of web references is created on the web server
and the configuration file specifying the different values of the parameters is created on the
client, which is then given as the input to \textit{surge}. \textit{Surge} uses the configuration file to generate
\textit{http} requests, to the specified server or servers, for a specified amount of time. The benchmark
outputs the average and variance of the number of requests received per second by the server.
Of the three benchmarks described above, although each generates http requests to the server, the workload they impose and consequently the throughputs they report are different. \textit{Netperf} is the basic test that exercises the network stack of the server, and gives an idea of the capacity of the server for the given request and response sizes. The throughput that \textit{netperf} reports is the maximum that the server can support without packet loss. \textit{Httperf} on the other hand reports the average throughput achieved over a period of time for the measured set of request-response samples. It also reports the minimum and maximum throughput measured over the interval. \textit{Httperf} reported metric tends to be optimistic as compared to \textit{netperf} since it is an average of the sampled measurements. It is necessary to set a sufficiently long measurement time interval so that any server mis-behaviors, due to saturation, is reflected in the output. \textit{Surge} is a benchmark that reproduces the bursty and self-similar nature of web traffic. Hence, the server throughput reported depicts this and is given as the average with variance. Generally, the average throughput reported has a large variance associated with it and the server CPU and other resource utilisations also reflect this behavior. The goal of this benchmark is to capture the bursty nature of the web traffic and thereby the increased resource usage during burst times as compared to the average requirement. \textit{Surge} is useful to understand the workload variation and the resulting demand for server resources. Of the three benchmarks, \textit{httpperf} gives a realistic behavior of the server under sustained loads. Hence, after the initial study of the three benchmarks, in terms of their behavior on virtualized servers, \textit{httpperf} is chosen to actually analyze the bottlenecks of the virtualization architecture in the thesis.

2.3.1 Case-Study 1: Virtualization effect on Application Performance

In this case-study the goal is to analyze the achievable performance of the chosen three benchmarks on \textit{Xen} and \textit{Vmware} virtualized servers. The metric used to denote the benchmark performance is the reply rate or throughput of the benchmark server in the case of \textit{netperf} and \textit{httpperf} and response time in the case of \textit{surge}. The reason why response time is chosen for \textit{surge} as against throughput is because of the way \textit{surge} is executed. \textit{Surge} configuration uses multiple parameters to capture the bursty nature of web usage but does not give a direct control
to vary the request rate and hence, the server throughput. As a result, to make a fair comparison, between non-virtualized, virtualized and the consolidated server, for the same workload configuration, the response time is chosen since it reflects the variation due to a difference in server environment.

Figure 2.5 captures the behavior of the benchmark performance, for the three described environments, when Xen is the virtualization technology used, on a single-core server. Initially, we analyze the performance on a single-core server since typical enterprise workloads contain a significant portion of I/O in their workloads because of which substantial part of CPU resource is free. Figure 2.5a depicts the change in maximum achievable throughput against the increase in response size. For the netperf benchmark, the loss of achievable throughput, for an application moving from non-virtualized to virtualized environment ranges from 4.0% to 20%. This behavior is because of the virtualization overheads. There is a further loss of throughput, in the range of 10% to 20%, when NIC sharing is added to virtualization, as is depicted in the case of consolidated server. The surprising result is the case of total throughput achievable across both the VMs on a consolidated server for netperf, which is represented by the "Total-Consolidated" graph. We observe that the total throughput achievable by the server 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 wherein 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 httpperf, Figure 2.5b, the improvement is not so dramatic. We observe here that while the reply rate increases linearly with the increase in request rate, until the peak rate, for the non-virtualized server, there is a gradual drop of throughput starting from 2% to 12.5% for the virtualized server, as we progress from the request rate of 500reqs/s to the peak rate. In the case of NIC sharing, for the consolidated server, the sustained peak throughput achievable by each VM is 450reqs/s and is less than half of what is achieved for the non-virtualized case. But, the sustained total throughput, including both the VMs, for the consolidated server improves to 800reqs/s. Yet, this still falls short of that achieved for the
Chapter 2. QoS and Virtualized Servers

Figure 2.5: Benchmark performance comparison charts for netperf, httperf and surge on non-virtualized, virtualized and consolidated single-core Xen server.

Non-virtualized server by 10%. This behavior clearly indicates that virtualization imposes overheads that cause loss of application performance as compared to non-virtualized environment. Thus the device that could support a given workload on a non-virtualized platform may not be able to support the same performance on a virtualized platform. A simple solution to this could be providing higher bandwidth device on the virtualized platform to support the same workload. Second observation is that, although there is a loss in achievable throughput per VM, the total server throughput is maintained reasonably well. This encourages consolidation of I/O workloads but with high bandwidth devices to support the same workload on the virtualized server.

The third figure, Figure 2.5c depicts the response times for surge benchmark. As compared
to the non-virtualized server there is an increase in response time for the requests in the case of virtualized server. The result for the consolidated server seems contradictory, since we observe that for a smaller number of requests the average response times are better than the virtualized server! However, as the number of requests increase, a significant rise in response time is observed for the consolidated server. This behavior is typical of the surge benchmark due to the inherent bursty nature of the generated workload. It is difficult to make a straightforward comparison for this benchmark because of the inherent variability imposed by the different parameters in generating the workload. The parameters are statistical estimates, that govern the bursty nature of the workload and usually lead to different access patterns by requesting different files at different times on the server. This change of access pattern causes a change in the resource requirements on the server and hence direct comparison across multiple benchmark runs, albeit with the same configuration file, may not be totally fair. We only use these results as a general observation, and rely more on the netperf and httperf results to understand the virtualization effect on the application, since these benchmarks can be configured to control workload that is repeatable across any number of executions. Furthermore, netperf is an idealistic workload because each transaction is generated only after the successful completion or failure of the previous one. The measurement is based on sequential transactions in a second. However, this is not true with real-life workloads wherein multiple, independent user requests arrive at the server at any moment. Hence, the server NIC would have a random load on its receive and transmit queues, which affects the server behavior and therefore, the application performance. Such behavior is exhibited by the httperf benchmark. Hence for the Vmware server we analyze the performance of only the httperf benchmark.

Figure 2.6 depicts the performance of the httperf benchmark on a single-core server where the virtualization technology used is Vmware-ESXi. Vmware-ESXi server implements NIC virtualization using device emulation. It can be observed that the overheads of emulation are comparatively quite high in relation to para-virtualization used in Xen. 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 in comparison to non-virtualized and para-virtualized systems. In this case 70% drop on the
maximum sustained throughput is observed when in comparison to the throughput achieved in the non-virtualized environment. This loss is visible even in the consolidated server case. The loss of throughput is due to increased CPU utilization, as is illustrated in case-study 2.

Hence, it is reasonable to believe that multi-cores 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 *http*-*perf* benchmark on the consolidated server with each VM pinned to a different core, for both *Xen* and *Vmware-ESXi* virtualized server show otherwise, as is indicated by the charts in Figure 2.7.

Application throughput increase is observed when in comparison to single core consolidated server, but this increase still falls short by 10% (950 replies/s as against 975 replies/s for non-virtualized) of what was achieved for the non-virtualized server. The reason for this shortcoming is because both VMs sharing the NIC also share the access path that is implemented by the 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 discussed in the next case-study.
Chapter 2. QoS and Virtualized Servers

2.3.2 Case-Study 2: Virtualization effect on Application Resource Usage

In Case-Study 1 we observed that virtualization does effect application performance and tends to degrade if the same server capacity is used for the virtualized server too. There is further degradation if consolidation with device sharing is also added to the virtualized server. In this case-study, we analyze why this degradation occurs. I/O device virtualization, implemented using software abstraction in prevalent technologies, adds one more layer of indirection. Because of this it is expected that resource requirement to support the same workload of the non-virtualized server would increase when hosted on the virtualized server. Figure 2.8 depicts three charts that show the server side CPU resource utilization for the performance observed in Figure 2.5. This figure contains three charts, one each for the benchmarks. Figure 2.8a shows the plot of %CPU utilization of the server against response message size for the netperf. And Figures 2.8b and 2.8c show the plot of server side %CPU utilization against request rate for benchmarks httperf and surge. In each of the charts 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 the application resource requirement. Further, when we look at the case where two-VMs are consolidated on a server, the server resource utilization increases significantly, as expected. On the consolidated server, each of the VMs sharing the same NIC, are also sharing the VMM and IDD. When compared to servicing a
Figure 2.8: Benchmark resource usage comparison charts for netperf, httperf and surge on non-virtualized, virtualized and consolidated Xen single-core server.

single VM, the VMM and IDD now have to also service the requests of the second VM. This increases the %CPU utilization on the server. Furthermore, the sharing of VMM and IDD by multiple VMs, adds to device access latencies, which contributes to a reduction in the maximum sustained throughput of the application.

This behavior is also observed in the case of httperf benchmark performance on Vmware server, as is depicted by the graphs in the chart of Figure 2.9. These tests were conducted on the Core 2 Duo server with two cores, instead of the Xeon server, because of hardware compatibility issues. Unlike the case of Xen, pinning of ESXi server (the hypervisor) to a CPU is not permitted by the hypervisor. Hence, any CPU utilization measurements for the ESXi hypervisor on Vmware show utilizations for all CPUs included. This results of CPU utilization
above 100% in the case of multi-core systems. Secondly, the Veam Monitor used for measuring the resource utilization does not allow resource usage data sampling beyond 1 second, which results in utilizations for the hypervisor to be coarse-grained. As a result, the graph for hypervisor utilization depicts an average for servicing requests for both the VMs. Thus while aggregating total server utilization, one needs to add up the utilization of one of the VMs and the hypervisor. Here also, virtualization of NIC results in using up more CPU to support network traffic on a VM when in comparison to a non-virtualized server. 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. We extend this study to the multi-core server

![Graph](image)

**Figure 2.9:** *httpperf* server %CPU utilization on *Vmware-ESXi* virtualized single-core server. The VMs are pinned to a single core while hypervisor uses all available cores

wherein each of the VMs and the hypervisor is pinned to an independent core. The resource utilization in this case is illustrated in Figure 2.10. In the Figure 2.10a 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 (see Figure 2.10a). 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 a concurrent device, supporting multiple VM 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 Figure 2.10b. In this figure we notice that the gross CPU utilization across the hypervisor and the VMs exceeds the total capacity of the server, which is counter-intuitive. (The ESXi server was installed on a two-core server.) This is because, the ESXi server does not have controls to pin the hypervisor to a CPU and hence the hypervisor uses up any of the available CPU resource. To account for this, while calculating the gross utilization for the system, we need to do an exclusive addition of either of the VMs CPU utilization with that of the hypervisor’s utilization. The rationale behind this is, when a VMM is using a CPU, the VM pinned to that CPU cannot be in execution. This however, is not the case with Xen hypervisor since it allows the hypervisor to be pinned to a CPU. All the Xen specific multi-core data was generated on a server with four-cores, and each of the VMs and the IDD were pinned to independent cores. Here also, we notice as in the case of single core experiments, the CPU Utilization by the hypervisor and the VMs is significantly much higher when 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 more VMs for sharing the NIC when compared to the Xen server.
2.3.3 Case-Study 3: QoS and Application Performance

The noteworthy point of observation from the above two case-studies is the behavior of each stream of benchmark in the case of consolidated server. We see that, in general, there is a further reduction in throughput on the consolidated server, when compared to the virtualized server, for both \textit{netperf} and \textit{httperf}, with a marked decrement in the later case. This indicates the obvious; lack of QoS controls would lead to severe interference in performance offered by the VMs that share I/O devices. In this case-study, we study the results of achievable performance, for a consolidated server that is hosting two VMs, using the prevailing mechanism for QoS control. The QoS controls are specific to monitoring and controlling I/O device usage by the VMs. Since we are analyzing the NIC sharing dynamics on a consolidated server, the QoS controls are associated with the sharing of the NIC resource, i.e. the network bandwidth that is made available to each of the VMs. Under normal circumstances, wherein no QoS controls are in place, the network bandwidth is shared between the VMs on a best effort, fair share basis. This is justified by the performance graph of the benchmarks on the consolidated server. On a consolidated server, addition of an extra VM sharing the NIC, it is expected that the achievable performance, of all the NIC sharing VMs, would decrease in proportion to the added workload. This is indeed observed for both \textit{Xen} and \textit{Vmware}.

The current commodity virtualization technologies like \textit{Xen} and \textit{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 \textit{Credit}, \textit{SEDF} or \textit{BVT} schedulers of \textit{Xen}. The CPU resource controls are fine-grained enough to deliver desired performance for CPU-based workloads [30, 52, 51, 94]. The problem is with I/O devices. The access to an I/O device is always through the hypervisor or the driver domain’s 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. But 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 Figure 2.11. To understand the effect of software based QoS controls for network bandwidth sharing, we present an experimental analysis of \texttt{httperf} benchmark on a consolidated server. The consolidated server hosts two VMs, namely VM1 and VM2, that are sharing a NIC. Each VM hosts one \texttt{http} server that responds to one \texttt{httperf} client. The \texttt{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, and VM2 is constrained by imposing a specific QoS value based on the desired application throughput. For both studies, each VM is subjected to equal load from its \texttt{httperf} client.

![Figure 2.11: Effect of hypervisor network bandwidth controls on application throughput for consolidated virtualized server hosting two VMs](image)

(a) \textit{Xen-IDD Linux} based QoS controls on VM2
(b) \textit{Vmware-ESXi} QoS controls on VM2
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 packet 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 its reception. Such controls are not available at present. Instead, in *Xen*, at-least one can use the *Linux netfilter* module’s network 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 suffering packet loss, by each of the VM is plotted in the graphs of Figures 2.11a and 2.11b. In these figures, the *x-axis* represents the *httpperf* request rate based on which the network bandwidth control was applied on the VM2 and the *y-axis* represents the maximum application throughput achieved by each of the VMs. In the case of *Xen*, *Linux tc* utility of the *netfilter* module is used to establish appropriate bandwidth controls. Specifically, each traffic stream from the VMs is segregated into different classes using *htb* traffic class with the VM2 stream controlled by a *tbf* queue discipline with the desired bandwidth control. 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.

Revisiting Figures 2.5b and 2.6, we observe that, on the consolidated server for the best effort sharing case, the NIC bandwidth 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 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, that VM1 performance
is expected to be better when in comparison to the best effort case. Figure 2.11 demonstrates that imposing QoS controls on VM2 does not translate to extra bandwidth availability for the unconstrained VM1.

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 in-spite of applying appropriate QoS controls. This makes the QoS guarantee weak. Because of which the usage of the device tends to be unrestricted even though the application experiences constraints. This can lead to denial of service attack on the device. To prevent this, the device should provide resource usage controls. In order to support such a model, it is essential to have micro-architecture support from the I/O device, which the existing devices lack.

On multicore servers, hosting many consolidated workloads of a data-center, indeterminate performance is definitely not acceptable. 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.

This is currently not the case for the resource controls for the network device in Xen or Vmware.

### 2.4 Issues with Prevalent NIC Virtualization Architectures

Current commodity multi-core technologies have system virtualization architectures that provide CPU workload isolation. The number of CPU-cores in comparison to I/O interfaces is high in multi-core 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 architect-ed 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 on to a multi-core 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 on isolated VMs) on the consolidated server, while sharing a single I/O device. Prevalent virtualization architectures suffer from several problems with regard to I/O device virtualization, as observed from the experimental results of the case-studies. These are:

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.

3. Lack of proper micro-architecture support from the I/O device, for resource usage control, makes the virtualization architecture susceptible to security lapses like denial of service attacks.

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. Based on our observations, we also notice that existing QoS controls for resource sharing are coarse grained. This is because these controls are implemented on architectures that are designed as an extension of single-hardware single-OS model to support single-hardware multiple-OS model. This amply confirms that it is indeed time to re-architect I/O device virtualization architectures.
2.5 Architecture wish-list to overcome existing lacunae

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. This limits application scalability for the same resource on the virtualized server.

- QoS features for regulating incoming and outgoing traffic are currently implemented in the software stack. Uncontrolled incoming traffic 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.

In subsequent chapters, a device virtualization architecture is proposed, described and evaluated. The proposal is an extension to I/O virtualization architecture, beyond what is recommended by the PCI-SIG IOV specification [68]. 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. High speed network devices, like 10Gbps 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.

2.6 Summary

In this chapter, we analyzed prevalent I/O device virtualization architectures in terms of the effect virtualization has on end-to-end application performance, on virtualized server resource utilization and effectiveness of existing QoS controls on resource allocation and usage control. We observe that the current architectures are an extension to the "single-OS single-hardware" model. This imposes severe restrictions in achievable application performance, usable device bandwidth and effectiveness of QoS controls on resource sharing with prevalent virtualization technologies. To overcome the deficiencies, we conclude that I/O devices on virtualized servers must support concurrent device usage among multiple VMs sharing the device and also enable resource usage QoS controls for finer control.
Chapter 3

Architecture for System Virtualization

The key driver for virtualization adoption in data-centers, apart from software isolation, is safe virtual machine performance isolation that can be achieved on 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 system virtualization architectures that provide CPU workload isolation. The number of CPU-cores, in comparison to I/O interfaces, is high in multi-core servers. This results in the sharing of I/O devices among independent virtual machines and hence, changes the I/O device sharing dynamics, when in comparison to, dedicated, non-virtualized servers.

On the unvirtualized servers, all the resources, like the processors, memory, I/O interfaces for disk and network access, are architect-ed 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. In contrast, when many independent applications are consolidated on to a multi-core server, using virtual machines, performance interference, caused due to shared resources, across multiple VMs, adds to the performance and security challenges.

The challenge is in ensuring performance of the independent I/O intensive applications, hosted inside individual VMs, on the consolidated server, while sharing a single I/O device.
Prevalent I/O virtualization architectures suffer from high overheads and performance interference due to device sharing. These issues cause variability in application performance that is dependent on the nature of consolidated workloads and the number of VMs sharing the I/O device. The performance interference also causes security vulnerabilities that lead to denial of service like attacks. One way to control this variability is to impose necessary QoS controls on resource allocation and usage of shared resources.

In this chapter, we highlight the requirement of resource provisioning constructs, from an application perspective, in section 3.1. This is followed by section 3.2, where identification of basic resource provisioning requirements, from system virtualization point of view, is carried out. We observe that, resources like CPU and memory satisfy the goals of system virtualization reasonably well. However, I/O devices like disks and network interfaces fall short of these goals, on several aspects. In section 3.3, we carry out the review of existing system virtualization technologies, to assess, if they meet the identified resource provisioning requirements. In section 3.4, we propose and describe NIC end-to-end virtualization architecture that addresses the shortcomings, and elaborate on the network packet work-flow using the architecture. In section 3.5, we conclude, highlighting the importance of light-weight design for system virtualization.

### 3.1 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 resources as and when required [88]. For the system to be able to do so, the application resource requirements are enumerated using resource requirement (RR) tuples. A 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:

\[ Application(RR) = \]

\[ (R1 < A1(Unit, Def, Min, Max), A2(Unit, Def, Min, Max), \ldots >, \]

\[ R2 < A1(Unit, Def, Min, Max), A2(Unit, Def, Min, Max), \ldots >, \]

\[ \ldots ) \]

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 [1], the following example in Figure 3.1 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 VM. Default value
Figure 3.1: An example Application Resource Requirement tuple for a VM, expressed in XML

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

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. And 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 VMM uses the resource tuples while allocating or deallocating resources to the VMs. It is assumed that RR contains values that are derived from the application’s performance requirements. In the context of multi-core servers, with server consolidation as the goal, each application is assumed to be hosted in an independent VM, which encapsulates the
application’s environment. Hence, the application’s resource tuples are 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 are arrived at, by aggregating the resource requirements of all the applications hosted by the VM.

There are several ways of arriving at the RR for an application that is to be hosted inside a VM. An expounded way is to profile the application for resource usage, along-with the system it is to be hosted on. Program profiling gives a variety of details about the dynamic behavior of the application in terms of the various functions, modules or libraries it uses for an invocation along-with the resource usage statistics. Furthermore, in many cases, a single application handles different use-cases, which could have varied resource usage patterns. Application resource profiling is an exhaustive mechanism for determining quite accurately the resource requirements of an application, for all its use-cases. Based on the profile data, one can easily construct the RR for the application.

While profiling gives accurate information, it is exhaustive and requires extensive efforts to generate and collate the data meaningfully. In practice, however, system resource utilization records can also give reasonably good information about the application’s resource requirement. This mechanism is easier and practical since it captures the requirements of those application use-cases that have been actually used. Moreover, since RR is constituted of max, min and default data of each resource, this mechanism is widely adopted and fairly easy to generate. From the data generated using either of methods explained, it is also possible to generate the resource usage trends. It is perceived that on virtualized servers hierarchical, resource specific, schedulers [66, 26] will play an important role in assuring performance guarantees. In such a situation, use-case specific, resource usage trends, can be effectively used by the prediction models of the resource schedulers. We do not pursue this topic further since it is outside the scope of this thesis.
3.2 System Virtualization Goals and Desirable Resource Provisioning Properties

The basic goal of system virtualization is to replicate complete, physical machine. This allows to host multiple, independent virtual machines on the same machine. Most widely adopted commercial end-use of system virtualization is server consolidation. Prevailing trends of system architecture will result in physical system resource sharing across independent virtual machines. This necessitates that the sharing of resources does not impose undue overheads and/or cause interference due to resource sharing on the VMs. As it is observed, from the experimental results of the previous chapter, existing technologies incorporate I/O virtualization architectures that are:

- heavy-weight in terms of resource usage and consequently, cause loss of usable device bandwidth,
- not scalable in terms of device sharing due to access path sharing, and
- provide coarse-grained resource usage QoS controls, that tend to be ineffective and inefficient in sharing the device.

This behavior mandates that I/O virtualization architectures need to be re-looked at, for the system design meeting the architecture goals. In the first survey paper on virtual machine research, [72], the author remarked that, the then existing system architectures, were not built for virtualization. Although the paper is dated, the relevance of the observation exists even for current day systems. In the paper, the author listed three desirable features for designing system hardware to support virtualization, which are:

- System hygiene: Virtual machines should not be built using instruction trap and simulation method since such approaches are clumsy and awkward.
- Software simplicity: Inherent hardware support for virtualization would make the VMM a small and simple program that further contributes to system reliability and security.
Chapter 3. Architecture for System Virtualization

- System performance: Systems that are designed to support virtualization should operate more efficiently than their non-virtualized counterparts.

To support such a design, the author suggested the idea of a hardware virtualizer. The hardware virtualizer, envisaged for virtual machines, was designed to support an instantaneous relationship between the resources of the virtual and real machine. In this design, all the resources of the real machine that need to support virtual resources need to be identified, and a map, called the f-map, between the resources of virtual and real machine, needs to be established. The functions distinctly associated with f-map are:

- The f-map transforms the virtual resource name to its corresponding real resource name.

- The f-map is invisible to all the software executing on the VM.

- All real and virtual resources are under the control of the VMM. VMM manipulates and invokes the f-map, and any f-map violation passes the control to the VMM.

- f-map is transparent to the VM in its state of execution in privileged or non-privileged mode. Hence, the VM executes as it would on a real machine.

The implications of these design principles, as claimed by the authors in [72], from resource management perspective, are those that meet the following properties:

1. Efficiency: Virtual Machine should be as efficient as the non-virtualized machine, if not more! However, one cannot expect virtualized systems to perform better that their non-virtualized counterparts. At the same time, the overheads must not be excessive so as to make virtualization prohibitive to use. Practically speaking, the virtualization overheads must be within reasonable limits so as to allow effective usage of system resources.

2. Isolation: Virtual Machine is an isolated duplicate of the real machine. Hence, sharing of resources on virtualized machine should cause minimal interference amongst the VMs.

3. Safety: The Virtual Machine Monitor should have complete control of resources. This implies that any VM cannot access any resource not explicitly allocated to it, and it is possible for the VMM to regain control of resources already allocated to any of the VMs.
To ensure that these properties prevail, the resource management constructs used for virtualization, should be light-weight and closer to real resources. This allows for tighter resource usage controls and provides for appropriate stubs for the much desired hierarchical adaptive scheduling framework for virtualized environments [26, 66]. Existing hardware design is architect-ed with the goal of allowing a single OS to use, control and manage all the resources. Most operating systems are process centric and system architecture is designed to benefit this model.

Prevalent general purpose operating systems project the physical or real hardware resources using OS abstractions like processes, pages, files, sockets, packets, etc. Sharing, of the physical resource, across multiple processes or tasks occurs, within the context of the OS, using these resource abstractions. Majority of the resource control and sharing policies, are built over the OS resource abstractions and are used by the resource schedulers within the OS.

System virtualization, on the other hand, by its very definition, requires every resource to support concurrency in terms of access context. Support for concurrency of resource usage for synchronous operation is different from support for asynchronous operation. Resources like the CPU, memory, internal and external caches are examples of resources supporting synchronous execution. Both memory and caches are synchronized based on the execution demands of the process that is currently using the CPU. In such resources, for process centric execution, the context information is copied into the context space of the resource, and execution is carried out. The process execution continues until the time-share allotted to it expires, wherein the process context information is saved into a specific location, usually the process address space, and the resource is allocated to the next eligible process. The process context information persists in the memory until the end of its execution.

For resources that support asynchronous operation, like the disks and NICs, the device has support for saving the context information associated with the abstract resource that the OS exposes over the device. In the case of disks the abstract resource is a file residing within the organization of a filesystem on the disk and for NIC it is a packet of a session connected to a socket of a process. In the prevailing system architectures the concurrency of access is supported at the file and socket abstraction, which is implemented through the OS kernel data
structures.

In traditional commodity systems, for resources like CPU, the physical system design is built to store the process context in hardware. This enables sharing of the resource by simply context switching the process. If the context switch overheads are kept minimal, time-sharing of the CPU resource gives fine-grained control on resource sharing, without losing out on its utilization. In the case of memory, a hierarchy of the processor caches hides the access latencies between the RAM and processor speeds. The cache and memory allocation units are carefully designed to ensure optimal memory use, without adding to the access latencies. Furthermore, application development tools aid to benefit, from the system memory hierarchies, by access strategies that reduce the cache misses. Apart from this, memory resource organization is designed to be concurrently accessed by different processes using hardware protection unit and translation buffer.

However, for the resources like I/O devices, the resource usage is supported using asynchronous mode of operation. This is basically to manage the difference in operational speeds of the I/O resources from CPU. Once the data from the I/O device is copied into the process address space, the process is initiated into execution by the scheduler. Since there is a limitation on the number of I/O devices that can be supported in the system, and also individual process may not really use up the total device bandwidth, the OS uses the same device to support many processes using its resource abstractions. A simple example is the NIC that caters to many communication streams within the same OS. The NIC is unaware of the streams which is managed by the network stack, supported within the OS. Because of the asynchronous nature of the device and the data handling associated with it, the OS maintains device specific memory buffers, called the I/O ring buffers. The OS handles the data transfers using these buffers and thus provides concurrent device access to multiple processes. These buffers are limited resources, and their organization, access and security are implementation specific to the OS. At the physical device, there is no notion of different users or processes and their access restrictions. These are managed by the OS at the resource abstraction level, in this case, the socket that a process has access to.
Prevalent commodity virtualization technologies, like Xen and Vmware, are built on system architectures, which are essentially designed for single OS access. Examining the features available for different resources with regard to the f-map functionality, we find that CPU resource meets most of the criteria. The virtual CPU (VCPU) mapping to the physical CPU is achieved through process schedulers. On the virtualized system, a VM is abstracted as a process inside the VMM and VCPU allocated to the VM is realized using time-sharing on a physical CPU. Since existing system designs support process context in the hardware, most of the CPU instructions from the VM can be executed directly, except for the privileged instructions. This yields good CPU performance for the VM. Hence, processor virtualization technologies, today, offer application performance close to non-virtualized servers [30, 52, 51, 94]. Memory, in prevalent technologies, is treated as an exclusive resource of a VM. Individual blocks of physical memory are exported to each VM and protected by the memory protection hardware. To enable flexibility, in available runtime memory to a VM, many of the existing virtualization technologies support dynamic memory expansion and reduction, which is achieved by the co-ordination of the VM and VMM, when initiated by the VMM. However, the memory hierarchy consisting of processor caches are still treated exclusive to the current process executing on the CPU. On multi-core platforms, executing applications using multiple cores, cache placement, particularly on shared cache processor architectures, becomes an important consideration. Specific cache placement policies are used to improve performance in general, but performance guarantees, associated with cache structure, size and placement, are yet to be addressed [21]. Processor architectures with shared caches could exhibit variable performance based on the workloads executing on the different cores of a CPU. It would be necessary to have fine-grained application specific cache control policies, directed by the VM, on such systems, to guarantee application performance. It is envisaged that a hierarchical cache control policy coordinated between the VM-VMM-hardware caches would guarantee and benefit the virtualized applications’ performance isolation.

The major drawback is with the I/O devices in these systems. There is no hardware support for concurrent access to multiple VMs on the I/O devices. As a result, the prevailing virtualization architectures support I/O device sharing, across many VMs, using software mechanisms.
The result is, device sharing includes the access path sharing also. Hence, serialization occurs at the device and within the software layers used to access the device. Thus extending this model to sharing of resources across multiple VMs has pitfalls which manifest as performance and security overheads.

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 file-system, that is exported to the VM. Any and every operation to this file-system, is from the VM. Moreover, all read and write disk operations are block operations. The data movements to and from the filesystem is synchronized using the file-system buffer cache that is resident within the VMs 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 based on the corresponding disk scheduler policies. 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 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 file-system policies get to be implemented within the software layers of the VMM or the hosting VM. Since the file-system activity is block based, setting up appropriate block sizes and with associated scheduling policies 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 who guarantee performance 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. Hence, sharing the 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 (in emulation) or address translation (in para-virtualization) due to virtualization. Moreover, since the device concurrency is presented by the VMM or the hosting VM, the scheduling policies are implemented higher in the network stack. This results in very coarse grained resource controls on the NIC, and hence, leads to the
ineffectiveness of bandwidth guarantees.

As observed in the previous chapter, the network device presents an interesting case to study virtualization architectures with performance isolation as the perspective. In both technologies, used for case-studies, since the virtualization techniques build over existing single-OS single-hardware model, there is a performance, resource utilization and security penalty. In the following sections of the chapter we present an end-to-end NIC virtualization architecture that addresses the identified issues. The I/O virtualization architecture is achieved using hardware defined reconfigurable virtual device interface (HDReconfig-VDI). In this thesis, NIC is identified as the I/O device for the case-studies to analyze the virtualization architectures, because it is a resource that has a strong potential for sharing, and higher utilization of such a device benefits many segments of users, specifically enterprise users. Also, as specified earlier, network device sharing leads to complex usage, and ensuring performance and security isolation is a major concern here.

In the context of the above discussion, we list here the desirable properties for the HDReconfig-VDI as:

1. Device hardware support for reconfigurable virtual device context.

2. Device time-sharing support by the device controller with variable time-slices to enable adaptive resource allocation.

3. Manifestation of virtual device over a physical device through late or dynamic-binding.

Supporting virtual device context, on a physical device, allows for native device access through the virtual device interface, and enables concurrency at the device. Concurrency on the device improves performance significantly. Native device access using the HDReconfig-VDI improves security of the virtualized server. Time-sharing over a resource improves its utilization and also system throughput for all the processes using the device. The biggest challenge to time-sharing is with the context switching time. If a device natively supports concurrency, context switch can be easily implemented in hardware by passing the control to the appropriate context that is ready for scheduling. This also improves the scope for better co-ordination between hierarchical device schedulers that are being proposed for virtualized
servers and virtual machine ecosystems [26, 66]. Furthermore, dynamic or late-binding creates a less-cohesive f-map that gives the flexibility to exploit unused resources and provide mechanisms for easy relocation and migration of VMs.

### 3.3 Review of I/O Virtualization Architectures

In the context of what is desirable, we review current technologies in this section, with specific reference to NIC virtualization. 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 [2] and para-virtualization as in Xen [24]. 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 OSes to execute inside the VM. Since it adopts instruction emulation, it tends to have high performance overheads, as was observed in the experimental studies described earlier. In Para-virtualization the Guest OS is also modified suitably to run concurrently with other VMs, on the same hardware. Hence, it is more efficient and offers lower performance overheads, when compared to emulation. 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 [77]. 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 [44, 80].
Recent publications on concurrent direct network access (CDNA) [97] and scalable self-virtualizing network interface [73, 14] are similar to the proposed work, in the sense that they explore the I/O virtualization issues on the multi-core platforms and provision for concurrent device access. However, the scalable self-virtualizing interface uses multi-core network interface. It uses separate cores for packet transfers and associated network processing. The authors do not detail how the device associated resource provisioning, security and address translation issues are handled, particularly in the case of virtualized environments. CDNA is architecturally closer to the our proposal, 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 access protection to these ring buffers is defined within the VMM, and hence, imposes constraints associated with resource utilization by the VMM. The VMM handles the virtual interrupts and the memory protection to share the I/O device. Moreover, 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 I/O virtualization architecture using HDReconfig-VDI addresses these and implements the basic constructs, to assign QoS attributes like required bandwidth and priority, in the device, to get fine-grained control on interference effects. This architecture has it basis in exokernels [33, 60] 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 our proposal, the extension of native device access is to the VM, with the protection being managed by the VMM and the device collectively. The hardware defined virtual device interface allows the VM to execute a traditional Guest OS, without any modifications, with native device drivers. This is a strong point in support of legacy environments without any need for code modification. Furthermore, 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 VMs, 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. Also, it does
not identify concurrent device access as a necessity for I/O device virtualization. Many implementations adhering to the IOV specification are now being introduced in the market by Intel [3], Neterion [7], NetXen [8], Solarflare [10], etc. Apart from these, the Crossbow [5] suite from SUN Microsystems talks about this kind of resource provisioning. However, Crossbow 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. 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.

Recent advances, in adopting virtualization, have also resulted in looking closely at the security issues with virtualization technologies on offer [48, 49, 84, 86]. Largely, it is believed that, the nature of security threats to virtualized systems is no different from the ones afflicting non-virtualized servers [35, 76]. What gets added, to the long list, is low-level hypervisor attacks and deployment of malicious virtual machines. Literature on system security has cited that faulty or malicious device drivers are one of the major causes of security threats. In this regard, prevalent I/O virtualization techniques have only increased this threat. I/O handling in traditional operating systems has always been through the privileged OS kernel. Prevalent virtualization architectures have merely extended these operating systems to support I/O device virtualization using a software layer. Such architectures force the device virtualization layer to be shared among all the VMs accessing the I/O device and thus make it the vulnerable component of the system. Any failures or crashes in this layer, which includes the device driver, leads to failure of all the VMs using the device. Both hosted and para-virtualized technologies are examples of such systems [48]. Furthermore, due to the software abstraction for virtual devices, these architectures are susceptible to denial of service attacks [35]. This is particularly due to unrestricted use of I/O device bandwidth on the incoming device queue [56]. Since on virtualized servers, single physical system is shared among multiple VMs, it is mandatory to have resource usage controls closer to the physical resource, rather than within the OS abstraction layers, as is the case with current technologies. Doing so leads to effective
control on monopolizing of the resource and limits denial of service attacks. We believe that virtualization can be used as a mechanism to provide the desired isolation [79] and security [19] if architect-ed correctly. For this, the hardware micro-architecture must be modified to support safe concurrent device access to the multiple VMs sharing it, and the virtualization architecture should permit the virtual devices to be defined within the VMs address space. With these inclusions, the proposed I/O virtualization architecture addresses the issues of vulnerabilities in the prevalent architectures.

3.4 End-to-end I/O Virtualization Architecture using Hardware Reconfigurable Virtual Device Interface

Based on the behavior of the benchmarks, described in chapter 2, the following bottlenecks are identified, for sharing network I/O device, among multiple VMs, on Xen or Vmware-ESXi virtualized server.

1. Virtualization imposes higher overheads for using I/O devices on virtualized platforms, which leads to increased CPU utilization of the hosting I/O Domain. Furthermore, if VMM and IDD do not support concurrent device access APIs, they themselves become the bottlenecks for sharing the device.

2. Virtualization overheads lead to loss of device bandwidth for a VM. Although, consolidation improves the overall device utilization among the multiple sharing VMs, the bandwidth loss forces the use of high-speed or higher bandwidth devices on virtualized platforms.

3. QoS features for regulating incoming and outgoing traffic are implemented within the network stack. Uncontrolled incoming traffic at the device to a VM, can severely impact the performance and security of other VMs sharing the device. This is because the decision to drop an incoming packet is taken after the device has received the packet. This could potentially cause non-deterministic performance and may lead to a denial of service attack on the VMs sharing the device.
To address these bottlenecks, we propose a device virtualization architecture that uses the HDReconfig-VDI. This proposal is an extension to I/O virtualization architecture, beyond what is recommended by the PCI-SIG IOV [4] specification. 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. High speed network devices, like 10Gbps 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 guaranteed application performance, 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 controls, for managing VM specific device usage. The architecture aims to reduce network I/O device access latency and enable improvement in effective usable bandwidth, in virtualized servers, 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 hosting I/O domain from the path of device access.
- Enable fine-grained resource usage controls at the device.

The analysis of prevalent commodity virtualization technologies, clearly highlights the issues that need to be addressed, while sharing I/O devices among independent VMs, on multi-core virtualized servers. It is also observed that, while para-virtualization offers better performance for the application, emulation is the alternative for improved consolidation. The goals are seemingly orthogonal, since current technologies build over virtualization unaware I/O devices. In our proposed architecture, we take 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 re-architecting the end-to-end virtualization architecture to deliver the benefits. In order to understand the benefits of this 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 (hosting I/O domain in Xen) 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 controls for managing the overheads, on a VM basis. Lack of such controls, leads to performance interference among the device sharing VMs.

- In Xen, 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 a drop in effective utilized device bandwidth. In multi-core servers with scarce I/O devices, this would mean having high-bandwidth, but underutilized devices, and low throughput applications on the consolidated server.

To overcome the above listed drawbacks, our 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 non-virtualized 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. Furthermore, this improves the application performance on virtualized servers and 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 Guest OS inside the VM, should use the NIC hardware that is enabled for QoS based concurrent device access.

![Figure 3.2: NIC virtualization architecture supporting multiple, independent, reconfigurable virtual-NICs](image)

**3.4.1 Proposed NIC Virtualization Architecture Description**

The NIC that supports the HDReconfig-VDI, is designed to meet the virtualization goals described earlier in this chapter. This NIC is an extension to a standard NIC that is defined by the PCI-SIG IOV specification. The NIC uses IO-MMU operations, as prescribed by the IOV specification, and also incorporates the virtual device interface (VDI). The VDI is a device context information that is shared between the physical device and the hypervisor and allows the use of the physical NIC by a VM, based on the specification of the context. The VDI defines a virtual NIC (vNIC) as a set of:
1. Device registers: A dedicated set of device registers is used by each of the vNIC to indicate the status of vNIC associated device operations. These could also be pre-designated address locations in the device memory partition allocated to the vNIC.

2. DMA Channel: It is proposed that the NIC supporting HDReconfig-VDI will have multiple DMA channels. Each vNIC will be allocated a DMA channel on request. Since at any given time, one vNIC would be using the associated DMA channel, it is represented as a dedicated link in the Figure 3.2.

3. Interrupt Line: The NIC will support message signaled interrupts (MSI). Using MSI allows each vNIC to be associated with an independent interrupt line, as designated in the Figure 3.2. This feature allows the hypervisor to decipher the device interrupt for the designated VM and generate the appropriate virtual interrupt for that VM.

4. Device memory partition: Each vNIC will be associated with a partition of the device memory. The partition’s association is identified with the vNIC and is populated into the hardware IOMMU by the VMM during the vNIC’s creation and initialization. The association of the partition’s context with the corresponding VM’s address space is maintained by the VMM and is shared with the corresponding VM, using the device address translation table, as depicted in Figure 3.2. The VM has read only access to this table. The device memory partition associated with the vNIC is mapped for I/O operations into the VM’s address space. Any change, that is initiated via the QoS management components of the architecture and resulting in the change of size of this partition, is effected by the VMM.

Figure 3.2 gives a block schematic of the proposed HDReconfig-VDI enabled NIC. 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

$$m \leq n$$
along-with device registers, form the virtual-NICs (vNICs). The device memory is reconfigurable, i.e. dynamically partition-able, and the VMs QoS requirements drive the sizing of the memory partition of a vNIC. The advantage of having a dynamically partition-able device memory is that any unused memory can be easily extended into or reduced from a vNIC in order to support adaptive QoS specifications. Apart from these, the NIC is built with additional functionality. It 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 its interrupt. 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:

Virtual-NIC: The virtual-NICs are enabled in hardware using the VDI interface. Since a VDI defines the independent access to the physical NIC, it supports concurrent native access to multiple VMs. This concurrency is achieved by using dynamic DMA channel allocation to a vNIC, MSI and dynamically partition-able device memory. These form the basic constructs to define a virtual device on a physical device as depicted in Figure 3.2. Each virtual device has a specific logical device address, like the MAC address in case of NICs, based on which the MSI is routed. The vNIC 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’s Guest OS to perform unconstrained DMA, and affect other VMs sharing the device, or the VMM itself. Furthermore, for high-speed NIC devices, the partition-able memory of the vNIC is useful in setting up large receive and segment offload capabilities that are specific to each vNIC, and thus customize the sizing of the vNIC based on the QoS requirements of the VM. Apart from these, the NIC controller can also be easily built with priority based processing for a vNIC as part of its time-sharing schedule.

**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 Guest OS 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. Moreover, 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 Guest OS of 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 devices address translation table, and does DMA directly into the VMs address space.

**QoS and virtual-NIC:** The virtual-NIC contains a portion of the device memory in its VDI which is carved out based on the device specifications that the virtual-NIC has to support. The device specification is derived based on the VMs QoS requirements as described in section 3.1. At the time of virtual-NIC initialization, the default unit of resource attribute is used to derive the memory partition size. This logic is typically incorporated in the NIC controller that supports QoS enablement in the architecture. Both the min and max attributes can be used by the NIC controller to increase or decrease the memory partition size, depending on the algorithm that is used to enable adaptive resource allocation.

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 translate 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 micro-architecture 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 Guest OS on a VM. Hence the security is verified and built outside the VM, i.e. within the device and regulated by the VMM. Allowing the 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 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 [56].

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

### 3.4.2 Network Packet work-flow using the Proposed NIC Virtualization 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 hosting domain. Only the device interrupts are routed through the VMM. In Figures 3.3a, and 3.3b, the work-flow 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 vNICs portion of the device memory and raises an interrupt. The VMM intercepts the interrupt, determines the destination VM, forwards the interrupt to the VM based on the vNIC’s priority. The device driver inside the VM issues DMA request. On completion of the DMA request, the VM’s network stack receives the data from the VM specific device ring buffers, as it would have done in the case of a non-virtualized server. In the case of transmission, the device driver that is resident in Guest OS of the VM, does a DMA transfer, of the data, directly into the vNIC’s mapped memory and sets the appropriate registers to initiate data transmission. The NIC transmits this data based on the vNICs properties like speed, bandwidth and priority. It may be noted that the code changes to support this architecture, in the existing implementation will be minimal. Each VM can use the native device driver for it’s vNIC. This device driver is the standard device driver for the IOV compliant devices, with the only difference that it can access only the restricted device address range. 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.

3.5 Summary

The basic aim of the future VMMs will be to maximize resource utilization without compromising on the VM’s performance or security. With a view towards such a goal, we have identified the lacunae in prevalent virtualization technologies and proposed the hardware defined, reconfigurable virtual device interface. The virtual device interface is designed to provide efficient and secure concurrent access to the VM’s sharing a device on the virtualized platform. We envisage that the virtual devices needed by the VM will be guided by the resource requirements of the VM. These resource requirements are derived based on the requirements of the applications hosted inside the VM. In this chapter, we described how an application’s resource
requirements are enumerated, using the resource requirement tuple. The hypervisor is expected
to use this tuple to allocate resources for the VM. The RR tuple is a structured description, of
all resources that the application, within the VM is affected by. We then described the desir-
able resource provisioning constructs, that virtualizable resources should support, using the
virtual device interface. Any virtualizable resource must support virtual resource context to
allow native access through the exposed virtual resource. Furthermore, the mapping of the
virtual resource to physical resource must be supported using dynamic binding to allow for
changeable resource binding. The resource sharing using the multiple virtual contexts must be
through time-sharing of the physical resource. The switching of the virtual contexts, should be
supported by the device controller with properties like priority, variable time-slices, etc. to en-
able hierarchical resource schedulers. Based on these guidelines, we proposed and described
the end-to-end NIC virtualization architecture. Further, the architecture realization and its use
is illustrated using the network packet work-flow.
Chapter 4

Modeling Virtual Machines using Layered Queuing Networks

This chapter deals with the issues of modeling end-to-end virtualization architectures. Virtualization adoption is becoming widespread and with the gain in using varied technologies, it is evident that there is a serious need for standard methods for modeling and evaluating these different technologies. Most efforts are currently directed towards identification as well as evaluation of suitable virtualization benchmarks [70, 71, 43]. This is because performance evaluation of diverse technologies is the major activity. Benchmark identification does help the user community to evaluate existing solutions. However, for evaluating architectures and changes made to the different layers of the virtualization stack, it is imperative to have a uniform framework for modeling multiple technologies to analyze their behavior. Modeling of virtualization environments can be complex, specifically if it involves end-to-end architectures. In this chapter, we present a survey of existing methods, available for modeling and evaluating virtualization architectures, with an aim to bring out the lacunae, from the perspective of end-to-end application performance. The basic requirement for virtualization environments is being able to capture the contention of software and device sharing, for concurrent use, which is critical for virtualization technologies. In view of this, we describe the layered queuing network (LQN) models involving rendezvous communication and illustrate how they satisfy the requirement. In the later sections of the chapter, we introduce the LQN software built by
Carleton University’s RADS laboratory. We then describe how LQN models are built, using this tool, to mimic existing virtualization architecture. For this we take the virtualized server built using Xen open source software as the example. Further, we present results for validating the LQN model, using the httperf benchmark, with that of experimentally observed results. We conclude the chapter by bringing out the assumptions used to build the LQN models and discuss their usefulness and limitations.

4.1 Introduction

The resurgence of virtualization is due to the availability of simple and effective solutions on commodity platforms. This has made virtualization a feasible solution to many small and large organizations. The efficacy and ease of use, of these solutions, has ensured a widespread adoption of virtualization. This usage has resulted in the need for evaluation of virtualization technologies for real life applications. Over this decade, many publications have brought to light the basic concerns that these virtualization technologies need to address [15, 30, 51, 77, 44, 49, 86, 35, 19, 95]. Of these, some of the bigger challenges are performance and security when system resources are shared on consolidated virtualized servers.

Virtualization technologies are built using varied mechanisms, to support virtual resources over physical ones. Hypothetically, one can view the virtualization mechanisms as a stack that we call the virtualization stack. This is akin to the architecture layers, depicted in chapter 1, to classify different virtualization methods. The virtualization stack can be visualized as a layered hierarchy of resource abstraction and management layers. This is depicted in Figure 4.1. The bottommost layer is composed of the actual physical resources that are abstracted out by the hardware abstraction layer (HAL). The hardware unit, for each of the resources, has its own controller which schedules and manages the use of the resource. The OS of the VMM or the hosted domain, that has direct control of the resource, exploits these controllers to use the resource efficiently. The controllers are not displayed in the diagram of Figure 4.1. Virtualized servers use the HAL to provide the resource abstraction of process, memory pages, disk blocks, network packets, etc. Using these abstractions and the device drivers of the physical resources,
the OS realizes the functions of the physical resource manager. The virtual resources are built
over the resource abstractions of the VMM/hosted domain, and are managed by the policies
built into the hypervisor or the hosted domain OS. These virtual resources are exported to the
Guest OS of the virtual machine. The Guest OS builds its resource abstractions over the virtual
resources and has its own resource management constructs and policies for them.

![Virtualization stack depicting various resource abstraction and management layers](image)

Figure 4.1: Virtualization stack depicting various resource abstraction and management layers

To perform, a realistic and comprehensive evaluation of various virtualization technologies,
it is necessary to understand the virtualization stack implementing the technology, and also, of
multiple resource abstraction and management policies, on the application hosted within a VM.
It is prudent to verify the efficacy and performance of an existing technology, by testing the
desired application performance, on a physical prototype. However, this becomes unrealistic if
the architecture involves changes to many or all layers of the virtualization stack. Particularly,
if the change deals with resource abstractions. In such cases, it becomes necessary to move to
4.2 Modeling End-to-End Virtualized Server Architecture

Most of the virtualization efforts, reported so far, have been physically implemented and verified for the fulfillment of their design goals. The reason that this approach has been followed, is because of the common approach of implementing virtualization as a software layer; providing the necessary resource abstractions over the physical resources. Most of the abstractions are hinged on the fact that the virtual resources can be realized, by judicious time-sharing, of the physical resource. This is true in the case of many of the virtualization technologies that are in existence today. However, as illustrated in the earlier chapters, every software layer adds to the overheads of performance, both in terms of latency and throughput, security, reliability and availability. Since one of the most widespread use of virtualization is in the server consolidation space, these overheads are deterrents to its adoption. Recent trends, to overcome the overheads, by provisioning bigger and faster devices, seem to offset the problem, but not resolve it. Hence a holistic design approach holds a better promise. Making the hardware aware of virtualization, yields the benefits, of providing a simple and efficient solution, to virtualization security and performance overheads. The issue with this approach is that of verification. A holistic approach would involve re-architecting the virtualization stack. This would involve changes to the hardware and the software layers of VMM and Guest OS, if necessary. Unless the physical prototype of the hardware is available, it is not possible to verify such a system. Complete system simulators like simics [12] are available but they do not offer the programmability to design new hardware and integrate into the simulated system for verifying with the existing or the modified OSes. All other system simulators offer modification and testing of only parts of the virtualization stack. Hence, using a full system simulator is not feasible to evaluate the proposed architecture.

Next, if we analyze the functional structure of the virtualization stack, it is basically composed of loosely coupled entities that communicate using synchronous or asynchronous mode. Specifically, the I/O device sharing is based on blocked communication with multiple requests.
to the device being queued. The queuing order can be in the order of arrival or based on the policy defined by the device scheduler, and processed one by one. Similar serialization exists, when the data is transferred between the physical device and the OS kernel data structures. Other components of the virtualization stack communicate using the asynchronous mode of communication. Thus, we can conceptualize the end-to-end virtualization architecture to be a queuing model with serialization at the device access software and the device.

The concurrency of different elements of the virtualization stack, and the serialization at the device and at its access software, can be modeled using the rendezvous communication (RNV) among servers. A queuing network composed of servers that are involved in RNV are named as stochastic rendezvous networks (SRNs).

Secondly, since the virtualization stack is built up of layers, it can be conceptualized by a layered queuing network (LQN) that has rendezvous communication among the shared software and devices, and asynchronous communication among other interacting stack elements. Using this as the basis of end-to-end architecture modeling, for virtualized systems, in the following sections we describe what SRNs are, how LQNs using SRNs can be analyzed, how to build such models from the work-flow of the end-to-end virtualization architecture and then present and compare results obtained from the simulation of LQN for Xen virtualization architecture with that of experimental observations. This structure of LQNs and its complete analysis is published in [100, 99] and is being reproduced here for continuity sake.

4.3 Layered Queuing Networks using Rendezvous Communication for Modeling Shared Devices and Software in Virtualized Servers

The Stochastic Rendezvous Network (SRN) model [99] is used for performance evaluation of concurrent systems using a rendezvous for communication between parallel processes or tasks. The SRN model evolved due to the need for performance evaluation of distributed systems based on client-server architectures using blocked mode of communication. SRNs are
basically queuing network models composed of servers supporting RNV. To model real life applications the queuing networks are arranged as LQNs made of interacting layers consisting of three types of servers, namely, clients, active and pure servers. In such setups the clients are identified as those that generate requests for service and hence are also called as reference servers. The clients make requests to the active servers or pure servers. Active servers are those that can receive requests and also generate requests, like a typical middle-ware server in the multi-tier application architecture. Pure servers receive only requests and hence, are used to model hardware resources. Typically, in LQN models it is the active servers who are involved in an RNV. The RNV is a blocked communication and any other request to the server, during the rendezvous, is queued. The SRN model is different from a general queuing network since it incorporates the performance effects of partly serialized and partly overlapped execution of the rendezvous communication. In the following sections we use the term SRN to refer to the queuing network model that is layered as described above.

The rendezvous is a term that was coined to capture concurrency feature introduced by programming languages like ADA and operating systems. The rendezvous achieves communication through messaging passing constructs like send-and-wait and receive-and-wait primitives. A rendezvous communication, abbreviated as RNV, is executed as follows:

1. The task requesting a rendezvous sends a message and waits for the reply.

2. On the acceptor task, if the receive instruction has already been executed, the task is blocked and waits for the request to arrive. As soon as the request arrives, the waiting task receives the message and reacts to it by executing the code designed as its response. This is called the first phase of execution of the receiving task. When this execution is complete, the receiver sends a reply to the requesting task. In the case when the receiver task has not yet executed the receive instruction, the message from the requester is queued until it does execute a suitable receive instruction. Messages from the queue are taken based on the queue discipline order, for example in a FIFO queue the messages are processed in the first-come-first-server basis.
Chapter 4. Modeling Virtual Machines using Layered Queuing Networks

3. The requester task is unblocked from the wait by the reply and may resume its processing. Subsequent to the reply of first phase of execution, the accepter task may also continue and execute further phases. These phases, also called as the post-rendezvous phases, are concurrent with the requester.

A program that is written using tasks as the unit of concurrency and rendezvous (RNVs) for intertask communication represents a network of RNVs. The SRN is a model for such class of programs and supports a simplified stochastic structure. Within the framework of SRN, tasks request RNVs with other tasks randomly and independently, according to some defined probabilities. On the server tasks, requests are queued and served according to predefined queuing order. The basic SRN is defined as a collection of interacting tasks, with each task having the following properties:

- a task executes independently with stochastic, independent and identically distributed (iid) execution times, of known distribution, for each execution phase.

- a task executes in a cycle of multiple phases, of which the first one is in rendezvous between accepting an RNV and releasing its requester.

- a task can request an RNV with other tasks based on randomly selected probabilities within any phase of its execution.

A typical example of SRN is depicted in the Figure 4.2. The figure portrays three distinct server types that can be found in an SRN, namely reference servers $S_i$, active servers $S_j$ and pure servers $S_k$.

- **Reference servers** are used to model clients who seek service and hence generate requests. In an SRN model, the reference server cannot accept requests but can have phases of execution, (called as service time), or idle time (also called think time) associated with request generation. A reference server is modeled as an infinite loop for generating the requests, based on its description in terms of its phases and their service and think times. As depicted in the figure, reference servers are referred by the index $i$ with a range from $1, \ldots, R$. 
- **Active servers** within an SRN is used to model services usually rendered by software. An active server accepts requests as a typical server and also generates requests like a client. Such servers participate in a RNV by accepting requests and also as clients by making requests. Active servers provide a simple abstraction for modeling layered or multi-tiered architectures. The active servers are referred by the index $j$ with range $R + 1, \ldots, C$.

- **Pure Servers** are the third category of servers, in SRNs. These are used to model servers who accept only requests. Hence, the hardware resources like CPU, disks, and network interfaces are modeled using pure servers. When pure servers need to be specifically indexed, we use $k$ for referring to them with a range of $C + 1, \ldots, N$. Otherwise, in general, these servers are aggregated with the active servers and referred to by the index $j$, collectively.

Within the framework of SRNs, a server can have multiple entries. Individual entry is used to model serial activity executed by the server. For modeling shared hardware device, all the different tasks sharing the device, are represented as independent entries of the representative pure server. Each entry of a server is also called task and it can have multiple phases. Although, in the figure, the active servers are represented by index $j$ and pure servers by the index $k$, in the following discussion they are collectively referred to by using the index $j$, as and when convenient. Under such cases $j$ takes values from $R + 1, \ldots, N$, to include both the active and pure servers. As represented in the schematic of the Figure 4.2, each of the servers $S_i, S_j$ or $S_k$ can have multiple entries $E_1, E_2, \ldots, E_n$. Each of the entry can have phases $P_1, P_2, \ldots, P_m$. The entries of a task are indexed using $u$ and the phases of the entry by $p$. The mean service time for each phase of an entry is represented by the variable $x_{u,p}$ where the index $u$ indicates the entry number, $p$ indicates the phase number of the entry and $i$ represents the server number in the SRN. This index $i$ takes values from $1, \ldots, R$ for reference servers and assumes the values of index $j$ for the non-reference servers. Depending on the context, and the need to distinguish between different types of servers, index $i$ is used for the set of all servers, or only reference servers.

As an example, random variables $x_{1,1}, x_{1,2}, \ldots, x_{1,m}$ represent the service times of the
phases, \( p = 1, 2, ..., m \), of entry 1 of server \( i \). Each entry of the server and each phase within an entry is serialized and share a common server message queue. In Figure 4.2, server \( S_i \) makes an RNV call in the phase \( P_1 \) of entry \( E_1 \), to entry \( E_1 \) of server \( S_j \). An RNV call is always made to the phase \( P_1 \) of an entry. This is a blocked send. The call is identified by the variable \( y \) with appropriate subscripts to represent the caller and the receiver. The subscript order is ”entry of caller” ”entry of receiver” followed by ”phase of the caller entry”. Here, \( y_{1,1,1} \) represents a call from entry \( E_1 \) of server \( S_i \), to entry \( E_1 \) of server \( S_j \), in phase \( P_1 \). \( E_1 \) of \( S_i \) waits for this call to finish before completing the phase and proceeding to the next phase. For this call, the RNV request of \( S_i \) is accepted by server \( S_j \), if a receive instruction has already been executed by \( P_1 \) of \( E_1 \) at \( S_j \), otherwise the message is queued at \( S_j \) till an appropriate receive instruction is executed. An RNV to the active or pure server is typically executed in the phase \( P_1 \) of the requested entry, and on completion, the reply is sent to the requester to unblock it. The subsequent phases of these servers are executed concurrently with the requesting server. Complete details of the SRN models can be found in [99].
Two important observations can be made about the SRN model, in terms of their suitability to model virtualized servers, namely:

1. Serialization on the active or pure servers is achieved by entries and their phases and a common server message queue. In virtualized environments with device sharing, this feature is useful in capturing the serialization occurring at the device and on its access path.

2. Servers only block on a RNV call to the active or pure servers. Other phases of the servers execute concurrently to each other. This feature allows to model the concurrency in consolidated virtualized servers when different VMs and the VMM are hosted on the same machine.

Our interest is specifically in analyzing the achievable throughputs, to assess the virtualization architecture, and the SRN model can be analyzed using mean value analysis (MVA) techniques, for a quick check, or simulation, for more accurate results. Using the SRN of Figure 4.2, we have the following two equations, for calculating throughput of the servers. The basic premise is that, the reference servers only generate requests, and hence, their throughputs are known once the server is completely described. For a given SRN, if the reference servers are numbered \( i = 1, 2, ..., R \) and their entries using \( u = 1, ..., n \), we define the throughput of an entry, also termed as reference throughput, as \( \mu_{ui} \). The reference throughput or the request generation rate of the entry \( u \), of reference server \( S_i \), is found by the total time it takes to cycle through its phases, which gives the following equation for \( \mu_{ui} \)

\[
\mu_{ui} = 1/\left( \sum_{p=1}^{m} x_{ui,p} \right)
\]  

(4.1)

where, \( x_{ui,p} \) is the mean service time for each phase including the delay for waiting and service for all its nested RNVs.

In order to analyze the SRN, we assume that the network is in a statistically steady state. This implies that the request rate is less than or equal to the service rate. Hence, the throughputs of the task entry interactions are interrelated by the synchronization imposed by the RNVs.
Taking into account this serialization, the traffic equations for the other servers, also called the non-reference servers, can be arrived at. Specifically, the throughputs are derived by knowing the number of calls made to the non-reference server tasks by every request generated by the tasks of each of the reference server. These calls are defined by the variable $y$ where $y_{u_i u_j p}$ represents the calls made in the phase $p$ of entry $E_{u_i}$ to the entry $E_{u_j}$. For example, if the entry $E_1$ of server $S_i$ makes two calls to entry $E_2$ of server $S_j$ in phase 1 of its execution, then $y_{1,2,1} = 2$. Hence, in terms of throughputs, for the non-reference servers $j = R + 1, ..., N$ we have

$$
\mu_{u_j} = \sum_{i=1}^{N} \sum_{u_i=1}^{n} \sum_{p=1}^{m} y_{u_i u_j p} \mu_{u_i p}
$$

If all the calls made by an entry of a task of the reference server $S_i$, including all its execution phases, to the entry $E_{u_j}$ of the non-reference server $S_j$, is represented by $Y_{u_i u_j}$, then

$$
\mu_{u_j} = \sum_{i=1}^{N} \sum_{u_i=1}^{n} Y_{u_i u_j} \mu_{u_i}
$$  \hspace{1cm} (4.2)

The set of traffic equations generated by 4.2 can be solved by Gaussian elimination to determine the $\mu_{u_j}$ for the non-reference servers as a linear combination of the known reference throughputs $\mu_{u_i}$ with coefficients $A_{u_j u_i}$. Hence, equation 4.2 can be re-written as

$$
\mu_{u_j} = \sum_{i=1}^{R} A_{u_j u_i} \mu_{u_i}
$$  \hspace{1cm} (4.3)

Intuitively, the coefficient $A_{u_j u_i}$, refers to the number of calls to the entry $E_{u_j}$ of server $S_j$ made by the entries of server $S_i$. It is assumed here that the SRNs have a communication graph, connecting the various servers by an acyclic graph. Although, some types of cyclic graphs can also be solved, details of such graphs and their analysis is described in [99].

### 4.3.1 Throughput bounds for SRNs

Throughput bounds, in a queuing network, give an idea about an application’s performance expectation for the given architecture and workload. The throughput bounds on the SRNs can
be found using two assumptions, namely:

- For the reference servers that drive the request rate, **no-contention bounds** are found by ignoring the waiting imposed by request serialization. This is the upper bound that the architecture could possibly support for the given workload.

- For the non-reference servers, **utilization bounds** are found from the fact that no single server can have utilization more than 100%. The utilization bound gives an idea about the possible bottleneck server in the queuing network representing the architecture. Thus, it gives an idea of the limitation an architecture can potentially have on the achievable application throughput.

We define:

\( x_{u,p} \): mean time to complete phase \( p \) of entry \( u \) of server \( i \) including contention and service due to RNVs.

\( s_{u,p} \): mean execution time of phase \( p \) of entry \( u \) of server \( i \).

\( \bar{x}_{u,p} \): mean total execution time to complete phase \( p \) of entry \( u \) of server \( i \), ignoring contention due to RNVs, but including the execution time in the nested RNVs.

It is trivial to show that \( \bar{x}_{u,p} \) is calculated using the following equation:

\[
\bar{x}_{u,p} = \sum_{u_j} y_{u,u_j} p \bar{x}_{u_j 1} + s_{u,p} \quad \text{(4.4)}
\]

This equation is solved recursively by traversing from server \( j = N \) to \( j = R + 1 \) in an acyclic graph. Furthermore, we define

\( z_{u,i} \): mean time to complete all phases of entry \( u \) of server \( i \), including waiting; this is the busy time of the server before it starts next request.

\( z_{u,i} = \sum_{p=1}^{m} x_{u,p} \), and the associated throughput \( f_{u,i} = 1/z_{u,i} \).

\( \bar{z}_{u,i} \): mean total execution time to complete all phases of the entry \( u \) of server \( i \), ignoring waiting.

\( \bar{z}_{u,i} = \sum_{p=1}^{m} \bar{x}_{u,p} \), and the associated throughput \( f_{u,i} = 1/\bar{z}_{u,i} \).
Then, based on the no contention bounds assumption, it follows that

\[ f_{ui} \leq \bar{f}_{ui} \]  

(4.5)

for all reference servers \( i = 1,2,...,R \).

And, the utilization bounds assumption leads to the following equation

\[ \sum_{i=1}^{R} A_{ui}f_{ui} \leq \bar{z}_{uj} \]  

(4.6)

for all non-reference servers \( j = R + 1,...,N \)

4.3.2 Throughput measurements in SRNs

Strength of SRNs is the ability to model large networks for performance evaluation. The approximate solution gives results in real time to complete the analysis and assess performance. This suits well for our purpose since at this stage we need an initial estimate of the architecture’s performance. In order to carry out the MVA we make certain assumptions on the SRNs. The requests generated by the reference server are modeled as Poisson arrivals. This is a reasonable assumption since for the httperf benchmark, the http requests generated have a constant average rate. The second assumption is with regard to the service time distribution of the non-reference servers. We perform the MVA using the exponential distribution parameters for the service time of the non-reference servers. The choice is based on the observation, using experimental data, that the mean service time per request on the http server is more or less constant for different httperf workloads. Moreover, on a virtualized server, several different components, their configuration and policies within the Guest OS and the VMM contribute to this service time. Further, the analysis of the distribution of service time of the different non-reference servers, needs careful identification of effective parameters from Guest OS and the VMM and a methodical study of the effect of these parameters on its distribution. This is outside the scope of this study.

The throughput calculation for the entries of reference and non-reference servers in an SRN
is given by the following equations:

\[
   f_{ui} = 1 / \sum_{p=1}^{m} x_{uip} \quad \text{where} \quad i = 1, 2, \ldots, R \tag{4.7}
\]

\[
   f_{uj} = \sum_{i=1}^{R} A_{uiuj} f_{ui} \quad \text{where} \quad j = R + 1, \ldots, N \tag{4.8}
\]

The mean service time \( x_{uip} \) of the phase \( p \) of the entry \( u \) of the reference server \( i \) is given by the equation

\[
   x_{uip} = s_{uip} + \sum_{uj} y_{uipuj} w_{uj} \tag{4.9}
\]

The summation bounds for the second term of the RHS refers to the calls made in phase \( p \) by entry \( u \) of server \( i \) to the entries of various non-reference servers referred by \( j \). The approximation in the SRN algorithm is in making an estimation on the mean service time \( x_u \) and the waiting time \( w_u \) due to the RNVs involved. The waiting time \( w_{uiuj} \) for the phase \( p \) of entry \( u \) of server \( i \) to complete the RNV to server \( j \) is the delay suffered by \( i \) due to queuing at server \( j \) and then waiting for the phase-1 service of entry \( u \) of server \( j \). Hence, if \( w_{quiuj} \) denotes the mean waiting time of the request from server \( i \) in the queue at server \( j \), the mean waiting time \( w_{uiuj} \) is given by the following equation

\[
   w_{uiuj} = w_{quiuj} + x_{uj1} \tag{4.10}
\]

Initially, the algorithm for solving SRN sets \( w_{uiuj} = x_{uj1} \), and service time \( x_{uip} = \bar{x}_{uip} \) using the bounds found by equation 4.4. And then iterates over equations 4.10 and 4.9 till the waiting time difference in successive iterations falls below a minimum threshold. The essence of the algorithm is to calculate the quantity \( w_{quiuj} \). To arrive at a logical value of this waiting time there are two scenarios under which a request from an entry of server \( i \) will wait in queue at server \( j \), namely:

- Server \( j \) is servicing a request from some entry of a server other than the server of this entry. Because of the very nature of the RNV, a rendezvous request from any entry of server \( i \) cannot arrive at server \( j \) while it has not replied to the earlier request. We define
the mean value for this type of waiting as \( w_{q,i,j}(1) \). The request from server \( i \) can arrive to find that there are other requests queued at the server \( j \) so in this case it has to wait till all the other requests ahead of it are serviced. Else, it can find that the queue is empty but the server busy.

- Server \( j \) is executing a later phase resulting from a previous RNV with any entry of server \( i \) when it makes a second request. The mean value for this type of waiting is defined by \( w_{q,i,j}(2) \).

Hence, the waiting time in queue for the request from an entry \( u \) of server \( i \) to entry \( u \) of server \( j \) is given by the equation

\[
 w_{q,u,i,j} = w_{q,u,i,j}(1) + w_{q,u,i,j}(2) \tag{4.11}
\]

As of this discussion, and also the LQN model used in simulations in the later part of this thesis, the server queues have been assumed to be following the FIFO discipline. To calculate the quantity \( w_{q,u,i,j} \) we analyze the situation at server \( j \) at the instant the request is made by the server \( i \). This is also called the "arrival instant" condition [99]. Figure 4.3 illustrates different arrival instant conditions possible at a SRN server. In the figure, the x-axis depicts the time-line of some random time interval on a SRN server and the y-axis depicts request arrivals during the interval under study. The SRN server is shown to have three phases of execution, of which the first phase is in RNV. Requests \( R_1 \) to \( R_5 \) are used in the picture to illustrate different arrival instant conditions at the server involved in RNV. Requests \( R_1 \) and \( R_2 \) depict the situation when the server is idle, just before a request arrives. This is termed as "Type I" arrival. As soon as the request arrives, the server enters a busy period, with the phase-1 execution, in response to the request. On completion of phase-1, the reply to the request is sent, which unblocks the requester. The SRN server continues with the execution of rest of the phases for the invoked entry. At this time, the requester is free to send another request, while the server is still busy executing the later phases of the previous request. Request \( R_4 \) in Figure 4.3 depicts this kind of arrival instant condition, when the requests \( R_3 \) and \( R_4 \) are from the same server \( i \). Such an arrival, from the same requester at the SRN server, is called the overtaking event [100]. It is possible that two subsequent requests are from different servers, in which case the earlier
request can be locked in phase-1 execution at the SRN server. This condition is illustrated by requests $R_2$ and $R_3$ in Figure 4.3. In either case, even though the server queue is empty, the later request incurs a waiting period, due to the busy period of server. This is termed as "Type-II" arrival. The other situation that a request can encounter on arrival is that it finds the server queue populated and hence incurs queue waiting time too. This is termed as "Type III" arrival and is illustrated by requests $R_5$ and $R_6$. The requests that arrive at an SRN server can be segregated as overtaking or non-overtaking requests depending on the source of the arriving request as compared to the source of the request in execution or in queue. We now derive the probabilities for the different arrival types, for the overtaking and non-overtaking requests.

**Waiting overheads of Non-Overtaking requests**

In the case of non-overtaking requests, the arriving request waits if it finds the server busy, optionally with some requests already queued ahead of it. This component of waiting is denoted by $w_{u,a_j}(1)$ in equation 4.11. To calculate this quantity the events of significance at the arrival instant are:
Chapter 4. Modeling Virtual Machines using Layered Queuing Networks

- **InService**($a_ib_j - c_id_jp$): Is the event when entry $d$ of server $j$ is busy executing phase $p$ for a request from entry $c$ of server $i$, when the request from entry $a$ of server $i$ makes a request to entry $b$ of server $j$. From the SRN constraint it is implicit that entries $a$, $b$ and $c$ belong to different servers and entries $b$ and $d$ belong to the same server. This the waiting involved in Type II arrival for requests from different servers. In such cases the SRN server $j$ can be executing any of the phases $p \geq 1$. The intuition for arriving at this waiting time is estimating the residual time (ResTime) of the current execution with the probability of the event that such an execution is occurring (Prob(InService)).

- **InQueue**($a_ib_j - c_d_j$): Is the event when the request from entry $a$ of server $i$ arrives at server $j$ for entry $b$ and finds that an earlier request from entry $c$ to entry $d$ is waiting in queue. This is a Type III arrival. To derive this waiting time the underlying concept is in estimating the probability of server $j$ being busy so as to cause this waiting (Prob(InQueue)).

Using the above listed waiting components, the waiting component $w_{q_{u_iu_j}}(1)$ can be expressed by the following equation

$$w_{q_{u_iu_j}}(1) = \sum_{\forall S(c)} \sum_{a} ((x_{d_j} Prob(InQueue(a_ib_j - c_d_j)) + \sum_{p_d=1}^{n} \text{ResTime}(a_ib_j - d_jp_d) \text{Prob}(InService(a_ib_j - c_d_jp_d))) $$

(4.12)

where, $x_{d_j}$ is the mean duration of entry $d$, including all it phases, on server $j$. Therefore, $x_{d_j} = \sum_{p=1}^{n} x_{d_j}p$. The terms $S(c)$ and $S(a_i,b_j)$ represent the servers of entries $c$, $a_i$ and $b_j$; term $p_d$ represents phases of entry $d$. It is essential to determine the probability of the two events, "InService" and "InQueue", for arriving at the waiting time, above. Both these quantities can be deduced based on the maximum utilization of the server in the SRN.

A naive calculation, without taking into consideration effects of RNV in SRNs, for the probability of the event "InQueue" is given by equation

$$\text{Prob}(InQueue(a_ib_j - c_d_j)) = \mu_{c_id_j}^{(a_ib_j)} w_{c_id_j} $$

(4.13)
The quantity $\mu_{c_id_j}^{(a_ib_j)}$ is the effective throughput of entry $d_j$ from entry $c_i$ because of requests from entry $a_i$ to entry $b_j$. The effective throughput is reduction in throughput due to task interlocking because of RNVs. Request flows from dependent tasks can cause send or receive interlocks. For example, in the diagram of Figure 4.4, when the Entry-A of Server-1 sends an RNV to Entry-C of Server-3, Entry-B cannot send to Entry-D since Entry-A will not send to Entry-B then. This is a send interlock. Similarly, when Entry-A is waiting for a reply from Entry-B, it will not send to Entry-C. This is a receive interlock. Complete explanation and derivation of the effective throughputs to include effects of interlocking are available in [100].

We reproduce the effective throughput equations due to interlocking below, for continuity sake. Let $\mu_{e_i}^{ILS}$ denote the send interlocking throughput of entry $a_i$ because of requests from entry $c_i$. Then, for send interlocking we have,

$$
\mu_{c_id_j}^{(a_ib_j)} = \sum_{e_i \in S(a_i)} \mu_{e_ic_i} - \mu_{c_id_j}^{ILS} - \mu_{c_id_j}^{ILS} y_{c_id_j} \tag{4.14}
$$

where, $\mu_{e_ic_i}$ is the throughput of entry $c_i$ because of requests from entry $e_i$. Similarly, let $\mu_{a_i}^{ILR}$ be the receive interlocking throughput of entry $c_i$ because of requests from entry $a_i$. Then, for
receive interlocking, we have,

\[ \mu^{ILR}_{a_i} = \sum_{e_i \in S(c_i)} \mu_{e_i} \]

\[ \mu_{c_i}^{(a_i b_j)} = \mu_{c_i} (1 - \mu^{ILR}_{a_i} y_{a_i b_j} 1/\mu_{a_i b_j}) \] (4.15)

The algorithm for calculating the effective throughput \( \mu_{c_i}^{(a_i b_j)} \) is to use equation 4.14 if entry \( c_i \) is send interlocked with entry \( a_i \), use equation 4.15 if entry \( a_i \) is receive interlocked with entry \( c_i \), else set \( \mu_{c_i}^{(a_i b_j)} = \mu_{c_i} \). Similarly, the probability of requests that can be found in queue at server \( j \), is affected by the SRN constraints and thus modifies the probability of the “InQueue” event in equation 4.13. We observe two conditions that affect the “InQueue” event, which are:

1. When request from server \( i \) arrives at server \( j \), it cannot find its earlier requests queued.
   
   If we define \( \text{Prob}(j \text{ is serving a request from } i) = U_{ij} \), the term \( U_{ij} \) is defined as the utilization at server \( j \) due to requests from server \( i \) including all its phases \( p = 1, \ldots, m \). The utilization notation gets modified to denote the usage due to requests from a particular phase accordingly as \( U_{ijp} \). Therefore the probability of non-overtaking events at server \( j \) is given by \( 1 - U_{ij} \). And the modifier is \( 1/(1 - U_{ij}) \) [100].

2. Other request that is queued at server \( j \) can only be a overtaking request or new request.
   
   Hence, this factor is \( (1 - U_{ij1}/(U_j - U_{k1})) \) [100].

Taking these two factors into consideration equation 4.13 gets modified as below,

\[ \text{Prob(InQueue}(a_i b_j - c_i d_j)) = \mu_{c_i}^{(a_i b_j)} w_{c_i d_j} (1/(1 - U_{ij}) (1 - U_{ij1}/(U_j - U_{k1}))) \] (4.16)

The next quantity to determine is the residual time (ResTime) and the probability of “In-Service” event for the non-overtaking requests. We first determine the probability of the “In-Service” event for non-overtaking requests. Let \( P_{OT}^{(a_i b_j)} \) define the probability of request from \( a_i \) overtaking the execution initiated by request \( c_i \) at server \( j \). This value can be calculated using equation 4.21 and is described in the next section. Then, probability for non-overtaking requests is found by \( (1 - P_{OT}^{(a_i b_j)}) \). And, the probability that the server \( j \) is not servicing a request
from server $i$ is given by $(1 - U_{ij})$. Hence,

$$\text{Prob}(\text{InService}(a,b_j - c_id_jp_d)) = \mu^{(a,b_i)}_{c_id_j}x_{d_jp}(1 - P_{OT}^{(a,b_i)})/(1 - U_{ij}) \quad (4.17)$$

The term $\text{ResTime}(a,b_j - d_jp_d)$ denotes the residual time taken for completion of request in service by entry $d$ executing in phase $p$ on server $j$ on the arrival of request from entry $a$ of server $i$ to entry $b$ of server $j$. In order to arrive at the value for "ResTime", we consider the entry phase on the server to be composed of a series of intervals, alternating between service by the server $S_j$ with nested RNV delays. Let the service rate for each of these phase sub-intervals be $\mu_j$. Assuming the service time distribution of these phase sub-intervals to be exponential, and also assuming exponential subintervals for task blocking on a RNV call, we can derive the value for "ResTime" using the renewal theory. This proof is given in [99, 100], which we reproduce here for continuity sake. On a request arrival, the residual service at server $j$ can be in any of the following two states:

- The residual service starts with an internal service by the server $j$. By virtue of exponential distribution of this service time, the expectation of the residual time is $x_{jp}$.

- The residual phase starts with a nested RNV by server $j$ to some server $k$. Because of this, the first phase sub-interval is the residual of the delay experienced by server $j$ at server $k$. This waiting has expectation of $w_{jk}$. After this waiting, an internal service begins and the remainder has the expectation of the full phase as above. Hence, in this case the residual time is $w_{jk} + x_{jp}$.

To arrive at the expected "ResTime" the residual times for each of the cases is a weighted addition, according to the law of total probability. The probability of each condition is taken as the mean total duration of the condition per phase of service, divided by the mean total phase time $x_{jp}$. This yields the following equation

$$\text{ResTime}(a,b_j - d_jp_d) = x_{jp}\left[\left(1 + \sum_k y_{jkp}\right)\mu_j^{-1}/x_{jp}\right] + \sum_k \left(w_{jk} + x_{jp}\right)\left[y_{jkp}w_{jk}/x_{jp}\right]$$
which reduces to the following equation by using equation 4.9

\[ \text{ResTime}(a_i b_j - d_j p_d) = x_{d_j p} + \sum_{\forall u_j} (y_{d_j u_j} w_{d_j u_j}^2) / x_{d_j p} + \sum_{v=p+1}^m x_{d_j v} \]  \hspace{1cm} (4.18)

Using equations, 4.18, 4.17 and 4.16 the waiting component \( w_{q_{u_i u_j}}(1) \) in equation 4.11 is computed.

**Waiting overheads of Overtaking requests**

The contribution to the current request’s waiting time, because of the overtaking of prior request, is denoted by \( w_{q_{u_i u_j}}(2) \) in equation 4.11. This component can be expressed by the following equation

\[ w_{q_{u_i u_j}}(2) = \sum_{\forall c_i \in S(a_i)} \sum_{\forall d_j \in S(b_j)} \sum_{p_d=2}^m \text{ResTime}(a_i b_j - d_j p_d) \text{Prob}(\text{InService}(a_i b_j - c_i d_j p_d)) \]  \hspace{1cm} (4.19)

For overtaking events, it is obvious that the arriving request cannot find the earlier request in queue for service. Hence, 4.19 composes only of the ”InService” event. The ”ResTime” component for this event is arrived at using equation 4.18, except that, for this event, the phases for consideration exclude the first phase of the entry. The probability for this event is given by the following equation

\[ \text{Prob}(\text{InService}(a_i b_j - c_i d_j p)) = \mu_{c_i d_j} \text{Prob}(\alpha_{d_j p}) / \mu_{i j} \]  \hspace{1cm} (4.20)

\( \text{Prob}(\alpha_p) \) in equation 4.20 refers to the probability that request from entry \( a \) to entry \( b \) overtakes the completion of all phases of entry \( d \) initiated by entry \( c \) and arrives at server \( j \) when it is busy executing phase \( p \) of entry \( d \). To arrive at \( \text{Prob}(\alpha_p) \) we consider SRN constraints. Since server \( i \) cannot find server \( j \) executing in phase 1 of any of its entries, because of the blocking in the RNV. Hence,

\[ \text{Prob}(\alpha_{d_j 1}) = 0 \]
For $p > 1$, the period from reply of server $j$ to server $i$ until the next request from server $i$ to server $j$, is a race between the next request, and the completion of the processing initiated by the previous request. Let $t$ be the mean delay from the reply to the following request. Then, as derived in [99], we have the following equations

$$t = (f_i Y_{ij})^{-1} - w_{ij}$$

$$\text{Prob}(\alpha_{djp}) = \text{Prob}(t < x_{d,j}2 + x_{d,j}3 + \ldots + x_{d,j}p)$$

Assuming exponential distributions for the delay $t$, and phase service times $x_{d,j}p$, we have

$$\text{Prob}(\alpha_{d,j,2}) = x_{d,j}2 / (x_{d,j}2 + t)$$

$$\text{Prob}(\alpha_{d,j,3}) = (1 - \text{Prob}(\alpha_{d,j,2}) \text{Prob}(\alpha_{d,j,3} | \alpha_{d,j,2}))$$

$$= (1 - \text{Prob}(\alpha_{d,j,2})) (x_{d,j}3 / (x_{d,j}3 + t))$$

Similarly, for phase $p$

$$\text{Prob}(\alpha_{d,j,p}) = (1 - \text{Prob}(\alpha_{d,j,2})(1 - \text{Prob}(\alpha_{d,j,3}) \ldots (1 - \text{Prob}(\alpha_{d,j,(p-1)})) (x_{d,j,p} / (x_{d,j,p} + t))$$

Using equation 4.20 we can now find the quantity $P_{OT}^{(ab)}$ required in equation 4.17. This is given by the following equation,

$$P_{OT}^{(a,b)} = \sum_{c_i \in S(a_i)} \sum_{d_j \in S(b_j)} \sum_{p > 1} \text{Prob(InService}(a_i, b_j, c_i, d_j, p)) \quad (4.21)$$

Hence, using equations 4.20 and 4.18 we can calculate the waiting component $w_{q_{u,u_j}}(2)$ of equations 4.19 and 4.11. This completes the derivation of the waiting time computation in SRNs.
4.3.3 SRN Algorithm to calculate server throughputs.

The definitions and the equations given in the previous sections provide a mathematical framework for calculating performance metrics in the SRN. The basic outline for calculating the throughputs is as follows:

1. The throughputs of the reference servers, \( f_{ui} \), the execution time of each phase of the server entries, \( s_{ui,p} \) and the number of visits from an entry to another \( y_{ui,uj}p \) are the inputs to the SRN.

2. Using the input values given, the contention delays \( w_{ui,uj} \) are calculated using equations 4.19, 4.12, 4.11 and 4.10. The delays and probabilities for "InService" and "InQueue" arrival instants are derived recursively for the RNVs for the active servers and the pure servers, starting in the reverse order from the pure servers. It is assumed that pure servers only accept calls and hence by knowing the number of calls to a pure server and the associated service time for each of the call, these probabilities are easily calculated. The contention delays are also calculated from the pure servers to active servers and reference servers.

3. By knowing the contention delays, the mean delay, \( w_{ui,uj} \) for the active and reference servers is generated.

4. Using the mean delay and execution time, the mean service time \( x_{ui,p} \) for each phase of entry \( u \) for the active and reference servers is generated using 4.9.

5. Using the mean service times, throughputs for the reference servers is calculated using equation 4.7.

6. Other entry throughputs are calculated using the reference server throughputs using equation 4.8.

As can be observed, from the step 2 to step 6, values from previous steps are used in the next step. For a fast analysis using MVA, the algorithm is initialized by using mean service time values ignoring contention, from 4.9 and then iterated till mean waiting time stabilizes to
below a threshold. However, for simulation setup, starting from the first request, the algorithm can be computed for the desired arrival rates of the reference servers.

4.4 Generating Layered Queuing Network Models

The proposed architecture involves changes in the hardware and system software accessing and using the hardware. In order to evaluate the end-to-end application performance based on these architecture-OS interaction changes, we need a setup to capture the architecture parameters in terms of the interacting component service times, type of interaction amongst the various components and the associated queuing delays when sharing software components and devices. Layered Queuing Network (LQN) models including RNVs are queuing network models that capture interactions between layered server components and also the contentions when multiple processes share a common software or device. LQN models are more generic than the SRN models in the sense that they capture the semantics of other type of communication also, between different servers. Model parameters within the LQNs include those of SRNs except that now the type of communication is also specified as RNV or asynchronous. An asynchronous call is similar to "receive and forward" call. Hence, for such calls, the contention is only at execution server and the requester is free as soon as it forwards the call. Servers involved in asynchronous communication can be analyzed using the standard queuing models like M/M/1.

The first step to generating the LQN is deriving the end-to-end architecture’s contention model. The contention model depicts the interaction of various application and system components along-with the shared software (modeled as active server) or device (modeled as pure server). This can be generated using the use-case of the application and its interaction with the system. The contention model helps in identification of the various architecture components and the mode of communication between these components. Figure 4.5 depicts the software and device contention model for the Xen virtualized server. In the picture, the application clients are the reference servers that are generating requests to their corresponding servers,
called "app servers". Each of the app server is hosted on a different VM. The Xen-VMM, Xen-IDD and the VMs represent the active servers since they accept as well as generate requests based on the service. The hardware resources, like the CPUs $P_0$, $P_1$, $P_2$ and $P_3$ and the network interface card NIC, used by the active servers are modeled as pure servers. The solid lines in the picture depict the request path from the reference servers to the non-reference servers and the dashed lines indicate the realization of the execution of service of the active servers on the pure servers. Multiple solid or dashed lines to a server indicate sharing and associated contention. The active servers are modeled with single or multiple entries. Multiple entries are used when the server has different service times associated with differing requests. The pure servers are modeled as multiple entry servers to represent time-sharing of the hardware resource.

Based on the contention model, the next step is to derive the LQN model. In our case, we use the network packet work-flow diagrams to depict the receive and transmit use-cases and derive the LQN model from these work-flows. Using the method of layers (MOL) [78], and as described for the SRNs in the previous section, on the LQN model, one can make performance estimates and studies of the modeled system. Various issues like, which server in the system is a bottleneck, what kind of throughput or response times are sustainable within the given architecture, what are the server and device utilizations for a given workload, etc., can be studied.
in detail. MOL for LQNs is similar to the algorithm of SRNs but is implemented using all the different elements of LQNs. Complete details on LQN modeling and MOL are given in [78]. As of this thesis, the LQNs software developed at the RADS lab of Carleton University was used to generate the LQN model for the existing and proposed virtualization end-to-end architecture. This software incorporates LQN modeling including RNVs for communication. The software has a Java based GUI called LQNDEF [37] that can be used to generate LQN models of the end-to-end system architecture. Input parameters for the queuing model include specification of arrival rates for the non-reference servers, number of visits to different reference servers and service times for different entries of the reference servers. An analytical solver called LQSIM [37] is integrated into the GUI to get a fast estimate on the performance metrics of interest. The model generated using the GUI can also be saved into a meta-file that is taken as the input to LQN simulation based solver called PARASRVN. Since it is a simulation based solver, PARASRVN [37] is more compute intensive and accurate compared to the analytical solver. All results presented in this thesis are generated using the simulation based solver.

4.5 LQN Model for End-to-end Xen Virtualized Server

In order to evaluate the proposed device virtualization architecture using the LQN model, we first generate the LQN model for the existing Xen architecture for httpperf benchmark behavior. The results obtained using the LQN model are then validated against the experimental data to understand the limitations of the modeling setup. The model elements like the active servers and their corresponding entries are carefully chosen such that they are usable in the proposed architecture LQN model. The proposed architecture’s LQN model is then generated by using relevant components from the Xen LQN model. Figure 4.6 is a diagrammatic representation of the LQN generated to capture the device and software sharing in the existing Xen architecture. LQNs allow for intuitive modeling of the system of interest from the interaction work-flow. For this study we generate the LQN model manually using the LQNDEF software. In the model each functional unit of the work-flow is modeled as a server and the interactions across these servers are modeled as synchronous (represents RNV in LQNDEF) or asynchronous.
Figure 4.6: LQN Model for consolidated Xen virtualized server supporting two independent httpperf streams

Legend:
1. Request: Generate httpperf request.
2. DMA_IN, DMA_OUT: NIC Device Driver calls for data DMA.
3. Timer, Tintr: Timer event and timer interrupt handler.
4. ISRIN, ISROUT: Interrupt service handlers.
6. recv_Pkt, Send_Pkt: Copy packet from/to application space to/from I/O ring buffer.
7. Send_Rep: Generate reply to received request.
communication links based on the actual implementation in the system. The server of the LQN is alternatively called task within LQNDEF.

Following three basic rules are used to generate the LQN model:

1. Synchronous communication is used where the requesting task services requests serially. For example, when the IDD/VM is receiving data from the device buffer into I/O ring buffer it is a synchronous and a blocked operation. This blocking causes queuing delays at the shared device or software and is captured using synchronous communication link in the LQN model.

2. In the case of multiple kernel entities, like device driver DMA transmit and receive calls, only one of them can be active at a time. We represent such software components as multiple entries within a task, to capture serialization. For example, between the timer and device interrupt, only one can be serviced at a time. Hence, these are modeled as different entries within the same task, in this case the task VMM_ISRN. Other servers that can execute independently are modeled as different tasks.

3. To capture the effect of server consolidation, multiple servers that are hosted on the same physical machine are bound to the same set of resources. For example when the Xen-hypervisor hosts two VMs on the same machine, each of the VM’s tasks and the hypervisor’s tasks are all implemented on the same processor resource. Similarly, when both VMs share a network interface, all the associated tasks are bound to the same NIC resource (Within the LQNDEF, this aspect is captured under the ”PROCESSOR” identifier.).

The LQN model follows the interaction work-flow and each task of the model represents receive or transmit component of the work-flow functional element. The generated LQN is modeled as an open queuing network model and performance is measured in terms of the maximum throughput achieved by the application serviced by each of the VMs. The LQN of Figure 4.6 includes the workflow steps of both reception and transmission of the packet because it models a complete request-to-reply sequence. Each task in the LQN is positioned within a layer and tasks on a layer are independent that do not make calls to each other. Since
we are using an open queueing network model, the first layer, Layer 1, contains two tasks namely, \textit{httpf1\_post} and \textit{httpf2\_post}. These tasks model the \textit{httpf} client nodes. Each of these tasks has one entry called \textit{Request 1} and \textit{Request 2}, respectively. Each of the entries represents an \textit{httpf} request generated by the respective \textit{httpf} client. Each client’s request is directed to a specific VM. Since the VMs are consolidated on the same server and share a NIC, both requests are received by the same NIC. This is indicated by the communications flowing out of the tasks in Layer 1 to the task \textit{NIC\_IN} of Layer 2. The task \textit{NIC\_IN} represents the reception of a packet by the NIC and is modeled using two entries \textit{DMA1\_IN} and \textit{DMA2\_IN}.

In reality, for the NIC used in \textit{Xen} experiments, the packets to either of the VM are not distinguishable by the NIC. However, in the LQN we model this distinction using two different entries of the task. This is done to trace the request from its source to destination. Since the entries belong to a single task, SRN modeling ensures serialization. This serialization represents device sharing by the VMs in the model.

The communication link between entries of Layer 1 and Layer 2 is asynchronous and hence the picture denotes an open arrow head on the link. LQNDEF uses open arrow heads for asynchronous communication and solid arrow heads for RNVs or synchronous communication. Once the packet reaches the NIC, the device raises an interrupt that is received by the VMM and forwarded as a virtual interrupt to the IDD. The NIC physical device driver, residing in the IDD, DMA’s the packet from the device memory into the IDD’s I/O ring buffers. This is modeled by the \textit{NIC\_IN} task in the LQN. The device driver forwards the packet to the ethernet bridge in the IDD which then sends the packet to the backend driver for the para-virtualized NIC. This is represented by \textit{ISRIN1} and \textit{ISRIN2} entries of the task \textit{VMM\_ISRN} (Layer 3) making RNV calls to entries \textit{Recv1\_Pkt} and \textit{Recv2\_Pkt} respectively of the task \textit{IDD\_Recv} (Layer 4). After the backend driver receives the packet, it identifies the destination VM and performs the packet forwarding operation to the corresponding VM’s front-end para-virtualized NIC driver. This involves remapping of the page containing packet from the IDD’s address space to the VM’s shared data space for the version of Xen used in experiments. This is represented by the asynchronous communication link from entries of task \textit{VMM\_ISRN} to entries of task \textit{IDD\_ForwC} (Layer 4). The three tasks \textit{VMM\_ISRN}, \textit{IDD\_Recv} and \textit{IDD\_ForwC} represent
device access path serialization.

After remapping the data to the VM’s shared space, the IDD requests VMM to raise the virtual interrupt to the corresponding VM for announcing the packet arrival. This is represented by the asynchronous call from entry of task `IDD_ForwC` to the entry of task `httpS1_Recv` in Layer 5. This entry represents request reception by the `http` server hosted inside VM1. After request reception, it is processed and a reply is generated for it. This is represented by the task `httpS1_Reply` (Layer 6). Again, the communication link between Layer 5 and Layer 6 task entries is asynchronous.

Once a reply is generated, the packet transmission action from the VM is initiated. Layers 7 to 10 represent the task and their entries involved in the packet transmission from a VM to its `httpperf` client. The reply generated by the `http` server of a VM is packetized and copied into the I/O ring buffer of Guest OS which is in the shared space of the VM and the VMM. The VM then raises a virtual interrupt which is received by the VMM and forwarded to the IDD. The IDD remaps the data page into its address space and transmits the packet out to the client using the device driver of NIC.

Various tasks along-with their entries and service times, used in the LQN model, are depicted in Table 4.1. Each task within the model is listed under the ”Task Name” column. The realization of the task on a hardware resource is listed under the column ”Processor”. Multiple tasks sharing a single hardware resource are realized on the same processor. This allows for accounting of the resource serialization due to device sharing in the LQN. The entry of a task is listed under the ”Entry Name” column and its service times, associated with each phase are listed under phase specific columns. In the LQN model, no more than two phases for any entry were necessary. Also, to distinguish between the effect of serialization at the VMM and the IDD of one VM’s request on another, these tasks are modeled as multi-entry tasks. For example, in the task ”VMM_ISRIN” the entry ”ISR1IN” models the interrupt raised by the NIC when a request to VM1 arrives and ”ISR2IN” for the interrupt for VM2’s request. In actual implementation these are handled by the same service routine. By separating such functions on a request basis the throughputs and utilization metrics are segregated based on VM specific workload. One element that is incorporated in the LQN model and not shown in the work-flow
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. This is illustrated in Table 4.1.

Table 4.1: Execution service time demands for task entries, with their phases and processor deployment, used in the LQN model of Xen Virtualized Server, hosting two VMs, for httpperf benchmark

<table>
<thead>
<tr>
<th>TASK NAME</th>
<th>ENTRY NAME</th>
<th>PROCESSOR</th>
<th>PHASE1</th>
<th>PHASE2</th>
</tr>
</thead>
<tbody>
<tr>
<td>httperf1_post</td>
<td>Request1</td>
<td>Client1</td>
<td>1.0000e-10</td>
<td>0</td>
</tr>
<tr>
<td>NIC_IN</td>
<td>DMA1_IN</td>
<td>GigNIC</td>
<td>9.2400e-05</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>DMA2_IN</td>
<td>GigNIC</td>
<td>9.2400e-05</td>
<td>0</td>
</tr>
<tr>
<td>VMM_ISRIN</td>
<td>ISRIN1</td>
<td>Core1</td>
<td>8.7904e-05</td>
<td>1.0000e-10</td>
</tr>
<tr>
<td></td>
<td>ISRIN2</td>
<td>Core1</td>
<td>8.7904e-05</td>
<td>1.0000e-10</td>
</tr>
<tr>
<td></td>
<td>TintrH</td>
<td>Core1</td>
<td>4.7783e-05</td>
<td>0</td>
</tr>
<tr>
<td>IDD_FwdC</td>
<td>Recv1_Pkt</td>
<td>Core1</td>
<td>2.9256e-05</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Recv2_Pkt</td>
<td>Core1</td>
<td>2.9256e-05</td>
<td>0</td>
</tr>
<tr>
<td>httpS1_Recv</td>
<td>Recv1_Req</td>
<td>Core1</td>
<td>3.0700e-04</td>
<td>0</td>
</tr>
<tr>
<td>httpS1_Reply</td>
<td>Send1_Rep</td>
<td>Core1</td>
<td>3.0700e-04</td>
<td>0</td>
</tr>
<tr>
<td>IDD_RevC</td>
<td>Rev1_Pkt</td>
<td>Core1</td>
<td>1.2500e-04</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Rev2_Pkt</td>
<td>Core1</td>
<td>1.2500e-04</td>
<td>0</td>
</tr>
<tr>
<td>IDD_Send</td>
<td>Send1_Pkt</td>
<td>Core1</td>
<td>2.9256e-05</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Send2_Pkt</td>
<td>Core1</td>
<td>2.9256e-05</td>
<td>0</td>
</tr>
<tr>
<td>VMM_ISROUT</td>
<td>ISR1OUT</td>
<td>Core1</td>
<td>1.0000e-10</td>
<td>8.7904e-05</td>
</tr>
<tr>
<td></td>
<td>ISR2OUT</td>
<td>Core1</td>
<td>1.0000e-10</td>
<td>8.7904e-05</td>
</tr>
<tr>
<td>NIC_OUT</td>
<td>DMA1_OUT</td>
<td>GigNIC</td>
<td>9.2400e-05</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>DMA2_OUT</td>
<td>GigNIC</td>
<td>9.2400e-05</td>
<td>0</td>
</tr>
<tr>
<td>httperf1_recv</td>
<td>Reply1</td>
<td>Client1</td>
<td>1.0000e-10</td>
<td>0</td>
</tr>
<tr>
<td>Timer</td>
<td>Timer_intr</td>
<td>Core1</td>
<td>1.0000e-10</td>
<td>0</td>
</tr>
<tr>
<td>IDD_Recv</td>
<td>Forw1_Pkt</td>
<td>Core1</td>
<td>1.2500e-04</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Forw2_Pkt</td>
<td>Core1</td>
<td>1.2500e-04</td>
<td>0</td>
</tr>
<tr>
<td>httperf2_post</td>
<td>Request2</td>
<td>Client2</td>
<td>1.0000e-10</td>
<td>0</td>
</tr>
<tr>
<td>httpS2_Recv</td>
<td>Recv2_Req</td>
<td>Core1</td>
<td>3.0700e-04</td>
<td>0</td>
</tr>
<tr>
<td>httpS2_Reply</td>
<td>Send2_Rep</td>
<td>Core1</td>
<td>3.0700e-04</td>
<td>0</td>
</tr>
<tr>
<td>httperf2_recv</td>
<td>Reply2</td>
<td>Client2</td>
<td>1.0000e-10</td>
<td>0</td>
</tr>
</tbody>
</table>

The service times of the entries are arrived at by using the xenoprof [64] kernel profiler and xenmon [38] resource monitoring tool on the Xen hypervisor and the hosted VMs. Xenoprof is an all-inclusive kernel profiler that allows to see the activity on each of the VMs and the
Xen hypervisor together. The profiler co-ordinates with its equivalent components located within each of the VMs and the hypervisor to give detailed statistical data on the use of system clock cycles, hardware events, etc. Using the clock cycle usage statistics for independent function calls of the Xen-hypervisor, and the VMs Guest OS and its applications, we derive the relative service times for each of the entries in the LQN model. It is to be noted that an entry of the LQN model is equivalent to a single or a group of functions that can be profiled by xenoprof. Depending on the context of the entry, these functions can be from within the hypervisor-OS, the Guest OS or the application running on the Guest OS. In order to execute Xenoprof, the hypervisor and the Guest OSes are booted with the profiler extended kernel. The profiler session is started before the benchmark executions are initiated and stopped after successful completion of the benchmark. This allows for the capture of data during the run of the benchmark. Since the benchmark is executed for a suitable time-period to provide steady state condition, the statistical data gathered can be used to generate average service time for the corresponding LQN entry. The service times so generated are relative, in the sense that they give details on how much percentage of the execution time was spent executing the entry.

To derive actual service times we use another tool called Xenmon. Xenmon is a comprehensive tool that captures resource utilization on a server, on a per VM-basis. Xenmon can generate detailed statistics on actual CPU usage by each of the VM for the given execution. Using the data generated by xenoprof and xenmon we can easily derive actual service times of each LQN entry for a given benchmark execution run. We normalize this data to derive the service times for a single http request. The data is generated by averaging service times for a request using multiple executions (about 5) for different workloads. It is observed that the service times calculated for a http request, thus, for different workloads is very close or identical. The service time values thus generated are populated onto the LQN model for the representative architecture.

All the service times specified are for a single http request and are measured in seconds. We use the httperf request rate to represent the mean arrival rate of the workload for the LQN. This assumption is made to simplify the LQN model; while in reality every http get request is broken into a sequence of packets that are passed through the various layers of OS, we
model it as a single service request. This assumption tends to give optimistic throughputs in the simulation results since it does not capture the packet queuing delays. Figure 4.7 depicts the performance measurements observed using the LQN model simulation for the Xen virtualization architecture. In each of the graphs, the total server throughput observed through experimental measurements is compared with that of the throughput obtained by LQN simulation. The X-axis denotes the \textit{httperf} workload and the Y-axis the achieved throughput. In the case of the consolidated server case the Y-axis represents the total throughput observed on the server. This is the sum of throughputs observed by each of the consolidated VMs. As can be observed from the validation graphs in Figure 4.7 for the application throughput, the deviation of the simulation results from the observed measurements is on an average less than 10% and thus can be used for evaluation of the proposed architecture. In each of the graphs of Figure 4.7 there are two curves, one representing the plot for experimental data, named ”measured”, and the other obtained by simulation, named ”simulated”, of the representative LQNs model. In each of the graphs the comparison is for the throughput metric. As can be observed from Figures 4.7a, 4.7b and 4.7c, LQN simulation results are almost in accordance with the experimentally observed values. The cut-off \textit{httperf} request rate for the simulated results corresponds to the stage where the server is saturated. The experimental data collected was for a single core machine and the LQN model is validated for such a system.

In order to understand the benefits of the existing Xen-virtualization architecture on a multi-core system, we extend the LQN model to a multi-core server by hosting the VMM/IDD and each of the VMs on an independent core. The expected difference between a single core and multi-core environment in terms of VMM overheads are VM context switching for the single core server which is accommodated in the service time differences for the associated task entries of the LQNS. On the multi-core virtualized server each core has to account for independent timer interrupt service which is handled by introducing the timer interrupt handler task. For modeling the multi-core virtualized server different processors are assigned to the tasks performing services of different VMs and the VMM. Table 4.2 depicts the task information for the multi-core LQN model. Here, \textit{core1}, \textit{core2} and \textit{core3} indicate distinct CPU cores of the multi-core virtualized system. As can be observed from the table, \textit{core1} is assigned to
Figure 4.7: \textit{httperf} Benchmark throughput validation charts for non-virtualized, virtualized and consolidated \textit{Xen} server. Comparison is between experimentally observed data with LQN model simulation results

the VMM and IDD tasks while \textit{core2} and \textit{core3} handle VM1 and VM2 processing requirements. The results for Xen multi-core virtualized server are presented in the Figure 4.7d. The graphs of this chart also indicate that the assumptions are reasonable to be used for architecture evaluation.

Similar graphs for server CPU utilization are presented in Figure 4.8. The graphs plotted are for a single core server. As can be observed from each of the Figures 4.8a, 4.8b and 4.8c the values obtained for server CPU Utilization through simulation of LQN model for \textit{Xen} architecture are close to experimentally observed values. The deviation of simulation values
Table 4.2: Processor-Task assignment used in the LQN model of Xen Multi-core Virtualized Server for *httpperf* benchmark

<table>
<thead>
<tr>
<th>Task Name</th>
<th>Type</th>
<th>Copies</th>
<th>Processor Name</th>
<th>Entry List</th>
</tr>
</thead>
<tbody>
<tr>
<td>httperf1_post</td>
<td>server</td>
<td>1</td>
<td>Client1</td>
<td>Request1</td>
</tr>
<tr>
<td>NIC_IN</td>
<td>server</td>
<td>1</td>
<td>GigNIC</td>
<td>DMA1_IN, DMA2_IN</td>
</tr>
<tr>
<td>VMM_ISRIN</td>
<td>server</td>
<td>1</td>
<td>core1</td>
<td>ISRIN1, ISRIN2, Tintr1H</td>
</tr>
<tr>
<td>IDD_Recv</td>
<td>server</td>
<td>1</td>
<td>core1</td>
<td>Recv1_Pkt, Recv2_Pkt</td>
</tr>
<tr>
<td>httpS1_Recv</td>
<td>server</td>
<td>1</td>
<td>core2</td>
<td>Recv1_Req</td>
</tr>
<tr>
<td>httpS1_Reply</td>
<td>server</td>
<td>1</td>
<td>core2</td>
<td>Send1_Rep</td>
</tr>
<tr>
<td>IDD_RevC</td>
<td>server</td>
<td>1</td>
<td>core1</td>
<td>Rev1_Pkt, Rev2_Pkt</td>
</tr>
<tr>
<td>IDD_Send</td>
<td>server</td>
<td>1</td>
<td>core1</td>
<td>Send1_Pkt, Send2_Pkt</td>
</tr>
<tr>
<td>VMM_ISRROUT</td>
<td>server</td>
<td>1</td>
<td>core1</td>
<td>ISR1OUT, ISR2OUT</td>
</tr>
<tr>
<td>NIC_OUT</td>
<td>server</td>
<td>1</td>
<td>GigNIC</td>
<td>DMA1_OUT, DMA2_OUT</td>
</tr>
<tr>
<td>httperf1.recv</td>
<td>server</td>
<td>1</td>
<td>Client1</td>
<td>Reply1</td>
</tr>
<tr>
<td>Timer1</td>
<td>server</td>
<td>1</td>
<td>core1</td>
<td>Timer1_intr</td>
</tr>
<tr>
<td>IDD_ForwC</td>
<td>server</td>
<td>1</td>
<td>core1</td>
<td>Forw1_Pkt, Forw2_Pkt</td>
</tr>
<tr>
<td>httperf2_post</td>
<td>server</td>
<td>1</td>
<td>Client2</td>
<td>Request2</td>
</tr>
<tr>
<td>httpS2_Recv</td>
<td>server</td>
<td>1</td>
<td>core3</td>
<td>Recv2_Req</td>
</tr>
<tr>
<td>httpS2_Reply</td>
<td>server</td>
<td>1</td>
<td>core3</td>
<td>Send2_Rep</td>
</tr>
<tr>
<td>httperf2_recv</td>
<td>server</td>
<td>1</td>
<td>Client2</td>
<td>Reply2</td>
</tr>
<tr>
<td>VM1_ISRIN</td>
<td>server</td>
<td>1</td>
<td>core2</td>
<td>Tintr2H</td>
</tr>
<tr>
<td>VM2_ISRIN</td>
<td>server</td>
<td>1</td>
<td>core3</td>
<td>Tintr3H</td>
</tr>
<tr>
<td>Timer2</td>
<td>server</td>
<td>1</td>
<td>core2</td>
<td>Timer2_intr</td>
</tr>
<tr>
<td>Timer3</td>
<td>server</td>
<td>1</td>
<td>core3</td>
<td>Timer3_intr</td>
</tr>
</tbody>
</table>
Chapter 4. Modeling Virtual Machines using Layered Queuing Networks

Figure 4.8: `httperf` Benchmark CPU Utilization validation charts for non-virtualized, virtualized and consolidated Xen server. Comparison is between experimentally observed data with LQN model simulation results from that of measured values, on an average is below 10%. The reason that the values donot tally as well as the throughput values is because of the limitations of the `xenoprof` profiler. This profiler allows to map resource utilization to the function or library call within the kernel. But the overheads due to the profiler using them is not distinguishable. As a result, the service times generated using the profiler statistics include these overheads, which leads to this variation. Also, we could not generate the CPU utilization values for the multi-core server specifically because the profiler does not give statistics on a per CPU basis for each of the calls. This separation is necessary particularly for the standard library calls which are used by the Xen
Chapter 4. Modeling Virtual Machines using Layered Queuing Networks

kernel as well as the http server and also the profiler. However, for the multi-core model we extrapolate the CPU Utilization on the basis of the throughput comparison graphs, since in the LQN the only thing that changes, for the multi-core model, is the association of a task to the processor. Rest all remain same.

![Figure 4.9: Simulation results for network bandwidth control modeling using server think time on Xen LQN Model](image)

Figure 4.9: Simulation results for network bandwidth control modeling using server think time on Xen LQN Model

The last comparison is for the results of imposing QoS on the LQN model. We use the think time parameter to model the delay imposed due to rate control of the netfilter module of Linux in Xen. The netfilter module has adaptive features for rate control. However, think time parameter is restrictive and generates predictable reply rate. Based on the generated reply rate the downstream behaviour of the model leads to the simulated application throughput. Figure 4.9 depicts the effect of network bandwidth control using the LQNs model of Xen. In the figure, VM1 supports unconstrained httpperf traffic and VM2 the constrained traffic. The x-axis represents the http throughput constraint imposed on VM2. The general trend of this graph is similar to what is observed experimentally (refer Figure 2.11a). However, comparison graph for the QoS study, between the experimental and simulated results, indicates higher achievable throughputs for both VMs on the experimental setup. This is because, the netfilter module of linux supports traffic shaping using qdisc and class constructs which are designed for adaptive rate control based on the traffic. What we ideally need is, to impose required bandwidth control on the traffic from VM2, leaving the unused bandwidth available to VM1. The closest that we can get to achieving this is by explicitly specifying bandwidth rate for the VM1 stream.
also. We set this limit to the maximum that the NIC can support. Such a specification is not necessary in the case of the LQN model since the model ensures flow separation by way of different tasks and their entries. The required bandwidth control is then easily imposed on the exact server and its entry as its think time. Details on how think time is suitable to model bandwidth limits on network flow is discussed in section 5.2.2.

Figure 4.10 gives the graphs for Xen server with QoS as VM1-M and VM2-M, and the graphs for LQN simulated results are depicted by VM1-S and VM2-S. As can be observed from the graphs in Figure 4.10, the experimental and simulated results match for controls beyond the point of saturation, where the behavior is similar to the best effort case. But, there is visible variation before this saturation for the measured values as compared to the simulated values. As explained earlier, this is due to the mechanism used for network bandwidth control. Linux netfilter module implements bandwidth control mechanisms using different control parameters like quantum of sharing for unused bandwidth, token bucket burst sizes, tolerable queue latencies, etc. For our experimental setup we are forced to use the higher quantum of bandwidth sharing than the default set by the netfilter in the Linux kernel due to its implementation issues. Linux kernel 2.6 requires the bandwidth sharing quantum to be set equal to the NIC’s MTU or higher. If the appropriate quantum is not set, the kernel issues warning messages, and causes erratic bandwidth allocation behaviour. However, increasing the bandwidth sharing quantum causes change in the achievable throughputs from the non-QoS server cases because this quantum dictates the extra tokens that the child class can borrow from its parent when it exceeds the set rate. Hence, we observe higher throughputs for the measured results when compared to simulated results. This is reflected in the deviation of the graphs VM1-M and VM2-M for measured values from that of simulated VM1-S and VM2-S graphs between the contraints of 350 to 650.

In the case of the simulation results, the LQN model achieves rate control by spacing requests to match the set rate. This mechanism is inflexible in terms of modeling the notion of excess bandwidth sharing. But it is precise in generating the average number of desired requests. On its downstream, because of the associated processing, the server in the LQN model suffers further bandwidth loss on the constrained channel, which is why it cannot achieve the
set rate. Hence, we observe lower achievable throughputs in the case of LQN simulation. However, both the graphs capture the fact that excess NIC bandwidth does not translate to higher application throughput. For evaluating the proposed architecture, we use the simulation results for comparing the expectation of the architecture when QoS constraints are imposed.

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 of throughput because of this.

- The experimental results for httpperf benchmark, illustrated in the thesis, 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 httpperf benchmark, the average arrival request rate is observed to be constant. Hence, the service times and arrival rates populated on the LQN model are modeled to be of exponential distribution.
• The service time for all device activities that are assumed to be executed in hardware and modeled as separate entries 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 experimental measurements made for the non-virtualized, virtualized and consolidated servers.

In general it is observed that the maximum throughput achieved using the LQN model is higher than the experimental results. The reason for this is because of the difference in the buffer sizes of the actual system and the LQN model. 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 3 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 non-virtualized 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.
4.6 Summary

Increasing adoption of virtualization technologies for enterprise applications has brought out the need for establishing a uniform framework for comparing different technologies. Standard benchmarks yield comparison data as perceived by end application. In order to compare and identify the performance bottlenecks on the virtualization stack, and how they can effect different applications it is necessary to have a testbed that helps in analyzing the various components and their associated behavior. We find that layered queuing networks are ideal for modeling such systems without losing out in details and at the same time being able to give reasonably good estimates on performance so as to enable evaluation of end-to-end architectures. The LQNs models enable sufficiently high level of abstraction of the system under consideration which enables using them for analyzing complex systems like virtualization stacks. The LQNs used in this thesis were built using LQNs software of Carleton University. Comparing the simulations with experimental observations we have found that they can be used to model and evaluate changes and enhancement on the virtualization stack with sufficient accuracy. It is however to be noted here that the models could be further enhanced for different arrival and service time distributions. Other issues like priority processing based on VM specific request, effect of scheduling discipline at the different hierarchy of virtualization stack, etc., are other interesting scenarios that can be analyzed using the LQNs.

In this chapter we have highlighted the need for a common framework for evaluating virtualization architectures and demonstrated the efficacy of LQNs in providing such a framework. We have identified RNVs as the necessary communication paradigm for modeling software and resource contention common to virtualization architectures. We described in brief the scope of RNVs as used in SRNs in analyzing the contention in concurrent systems. We also described the use of such communication primitives in LQNs and how these models can be analyzed to make performance estimates on the system behavior. Further, we listed out the basic assumptions made and the method for generating the LQNs model of the Xen virtualized server. The chapter ends with a comparison of throughput results obtained using simulations on the LQNs with that of experimental results. We find that LQNs are easy to build and accurate enough for evaluating virtualization system architectures.
Chapter 5

Evaluation of Proposed End-to-end Virtualization Architecture

The chapter describes the generation of an LQN model for the proposed architecture and the assumptions made in doing so. Since the proposed architecture involves changes in hardware and elements of the virtualization stack, it is necessary to understand how the service times are derived for such models. Getting an insight into these details gives us the knowledge to assess the accuracy and reliability of the architecture simulation results. The results of the proposed architecture are presented systematically for the three identified metrics namely, throughput, server CPU utilization and effectiveness of resource usage specific QoS controls. In conclusion we highlight the need and benefits of the architecture citing limitations on using LQNs for evaluating the architecture. The chapter ends with a summary of its contents.

5.1 Introduction

The analysis of prevalent commodity virtualization technologies in Chapter 2 clearly highlights the issues that need to be addressed while sharing I/O devices across independent VMs on multi-core virtualized servers. It is also observed that while para-virtualization offers better
performance for the application, emulation is the alternative technology 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 re-architecting the end-to-end virtualization architecture to deliver the benefits. In order to understand the benefits of the proposed architecture, 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:

1. 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.

2. 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 controls for managing the overheads on a per VM basis. Lack of such controls lead to performance interference in the device sharing VMs.

3. 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 multi-core servers with scarce I/O devices, this would mean having high bandwidth underutilized 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 non-virtualized 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.

5.2 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 VMs Guest OS, evaluation of the architecture is carried out using simulation on the LQN model of the end-to-end architecture. In LQN models, functional components of the architecture work-flow are represented as server entries. Service of each entry is rendered on a resource. End-to-end work-flow 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. One, there is a lack of appropriate system simulation tools that allow incorporating design of new hardware along-with VMM and Guest OS 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 work-flow, right from the application to the device including the intermediate layers of the virtualization stack. 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 as illustrated in Chapter 4.

5.2.1 LQN model for the proposed architecture

LQN models can be generated based on the network packet receive and transmit work-flows, manually, using the LQNDEF software developed at the RADS laboratory of Carleton University. In this chapter, results generated for the LQN model corresponding to the httpperf benchmark are presented for analysis, since the bottleneck issues are prominent for this benchmark. The LQN model generated for the proposed end-to-end architecture confirms to the
three assumptions listed out in the last chapter.

Figure 5.1 depicts the LQN model generated and used for simulation of the proposed NIC virtualization architecture on a consolidated server hosting two VMs, each supporting an independent httpperf client. Each functional element of the network packet’s receive/transmit work-flow (Figure 3.3) is represented as an entry of a task or server in the LQN. This LQN model is constructed on similar lines as that of the Xen LQN model described in the previous chapter. However, there are two distinguishable features that are different in the LQN model for the proposed architecture when compared to the Xen’s LQN model. Firstly, in the proposed architecture that is using the HDReconfig-VDI, native access to the VM is enabled. This enabled by using independent device partition to each VM. But since in NICs supporting multiple DMA channels only one DMA action can be active at a time device serialization still occurs. Hence, in Layer 2 of the picture, the two entries DMA1.IN and DMA2.IN are included in the task NIC.IN. And, because of this the IDD component entries (Layer 4) of Figure 4.6 are missing in the proposed architecture’s LQN. This eliminates the serialization on the device access path. As a result, all the activities associated with device access are now handled by independent tasks with single entries, as represented in Layer 4 and Layer 5 for packet reception and Layer 7 and Layer 8 for packet transmission, in Figure 5.1. Hence, we notice that the serialization in the proposed architecture is restricted to the device access entries of tasks NIC.IN and NIC.OUT. Other than these changes, all other entries, and their associations remain same in the LQN for the proposed architecture. The legend associated with the picture in Figure 5.1 gives a brief description of the functional element of the work-flow which the entry of the task is representing. The service times of the entries also remain same, except for the entries that identify and forward the packet to its destination VM, Forw_Pkt and Rev_Pkt. In the proposed architecture, these functions are part of the device and hence their service times are made equivalent to the device service times (10^{-10} seconds).

In general, it is observed that the maximum throughput achieved using the LQN model is higher than the experimental results. The reason for this the difference in the buffer sizes of the actual system and the LQN model. In the simulation environment, the buffer sizes used were much larger than the experimental system. This was deliberately chosen to ensure
Chapter 5. Evaluation of Proposed End-to-end Virtualization Architecture

Figure 5.1: Layered Queuing Network Model for the Proposed end-to-end NIC Virtualization Architecture for Consolidated Virtualized Server hosting two independent httpperf streams

Legend:
1. Request: Generate httpperf request.
2. DMA_IN, DMA_OUT: NIC Device Driver calls for data DMA.
3. Timer, Trlnr: Timer event and timer interrupt handler.
4. ISRIN, ISROUT: Interrupt service handlers.
6. Rev_Pkt, Send_Pkt: copy packet from or to application space to/from I/O ring buffer.
7. Send_Rep: Generate reply to received request.
that the architecture limitations are not a result of insufficient buffer sizes. Because of this, comparing simulation results with the experimental results is incorrect. In-order to make a fair comparison, we use of LQN simulation results of the Xen architecture with that of the LQN results for the proposed I/O virtualization architecture. For this 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 for Xen. After this, all comparisons for the LQN simulation results of the proposed architecture are made using the LQN simulation results of the Xen architecture rather than the experimental results.

5.2.2 Simulation and Results

The proposed architecture is evaluated using the results obtained by the parasrvm simulator of the \textit{LQNS} software package [37]. The architecture is evaluated for multi-core 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 multi-core server is carried out. Figure 5.2 depicts the results of achievable throughput and server CPU utilization for a multi-core Xen 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 Figures 5.2a and 5.2b, in a multi-core environment with Xen-IDD, VM1 and VM2 each pinned to a different core, and each VM servicing one \textit{httpperf} stream, the maximum throughput, without loss, achievable per stream is 950 requests/s as against, 450 requests/s in the case of single-core virtualized server. But, for the maximum throughput, the CPU utilization saturates for the Xen-IDD, which is hosting the NIC of the server. This indicates that further increase in application throughput is impossible since the processor core serving the Xen-IDD has no computing power left. Figure 5.3 shows these statistics for a similar situation but with the proposed I/O virtualization architecture. We observe from Figures 5.3a and 5.3b, the maximum throughput achievable 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% when in comparison to what was achieved on the existing *Xen* architecture.

![Graphs](image)

(a) Throughput  
(b) CPU Utilization

Figure 5.2: Maximum throughput achievable per *httperf* stream and CPU utilization for existing *Xen* architecture on a multi-core server hosting two VMs each servicing one *httperf* client. The IDD, VM1 and VM2 are pinned to independent cores

Also, from Figure 5.3b it is observed that the CPU Utilization of the *IDD* or the hypervisor has considerably reduced. The reason for this behavior is that, the NIC is now handling the identity of the packet destination. Also, in the Xen model, bridging software that 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 modeling assumptions are made to simulate the network bandwidth controls implemented in the *netfilter* module of
Chapter 5. Evaluation of Proposed End-to-end Virtualization Architecture

(a) Throughput

(b) CPU Utilization

Figure 5.3: Maximum achievable throughput and CPU utilization charts for a multi-core virtualized server incorporating the proposed I/O virtualization architecture and hosting two VMs, pinned to different cores, each servicing one httpperf stream

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 work-flow. 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. The think time parameter is used to model bandwidth restriction in the LQN model. 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. But within Linux there is a parameter to allow bursts of packets above the imposed bandwidth constraint. This is the burst parameter of the bandwidth control mechanism in the 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. But, 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 [55].

The following graphs in Figure 5.4 depict the effect of not imposing (Figure 5.4a) and imposing network bandwidth QoS controls on the incoming stream of VM2 (Figure 5.4b), 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 Figure 5.4a, 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 Figure 5.4b 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 the NIC is discarding requests to VM2 that are above the specified request rate, VM1 can use the available bandwidth, 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 towards each other and finally merge into 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 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 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 [6]. 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. Since, now the interference of device usage is eliminated, the threat of service denial attacks arising out of this interference is also eliminated.

![Graph](image.png)

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

Figure 5.4: NIC QoS effect on Application Throughput for the Proposed Architecture

5.3 Summary

In the chapter we described the procedure for generating the LQN model for an end-to-end architecture that is composed by integrating different elements of the existing architecture. Such methods are essential in situations where one is exploring the design space to meet specific
objectives. Such scenarios mandate a simple, intuitive and easy to generate models that can be evaluated without much restraints. The methods adopted should enable capturing essential system abstractions within reasonable assumptions and be able to generate results that can be used to evaluate the design. As described in the chapter, LQN models actually give the benefit of simplicity and accuracy to enable this evaluation. However, it is to be noted that the software used for analysis has some limitations like limited type of distributions for arrival rates and service times. In order to realistically model real-life applications it will be necessary to extend these to include other queuing models like $M/G/m/B$, $G/G/m/B$, etc. Also, in order to make packet level simulation realistic in the LQN model, it is not only necessary to allow generation of multiple requests from an entry but also allow merging of multiple requests on an entry and be able to associate that with a single execution. Based on the observed results we can get the upper bounds on the server throughputs which is an important result since that indicates clearly the benefit or deterioration of a proposed architecture. As indicated by the observed results we find a clear demonstration of the benefit, in quantitative terms, of the proposed architecture for all the three cases indicated in Chapter 2.

In summary, this chapter details the procedure for capturing end-to-end architectural changes using a LQN model. The necessary assumptions made to enable this modeling are described to give an insight into understanding the observed results. Further, results obtained by simulation on the LQN model for all the identified cases of the benchmark are presented and analyzed systematically to understand if the proposed architecture could potentially yield the promised benefits. On the existing technologies for I/O workloads, we observe that the application throughput on a VM is reduced and server CPU utilization is increased when moved from a non-virtualized server to a virtualized server. This is attributed to the high virtualization overheads caused by the software layers used for providing I/O virtualization. Making the hardware device virtualization aware enables the I/O device for native access by a VM, and hence these overheads are considerably reduced. The throughput improvement is about 60% for the proposed architecture, with an equivalent reduction in CPU utilization due to virtualization. Also, enabling independent virtual device context on the hardware improves resource
usage specific QoS controls which is reflected in the observed application throughput. Furthermore, involving the physical device in access management of the virtual device provides better security management and hence reduces vulnerabilities associated with sharing the I/O device. Thus the proposed architecture improves performance and resolves identified security lacunae.
Chapter 6

Conclusions and Future Work

In this chapter, we summarize the content of the thesis and reinforce the need to re-architect I/O devices for virtualization. The thesis discusses and evaluates one new method of I/O virtualization. The future work section of the chapter lists out some interesting other possibilities that hold promise and can be explored.

6.1 Conclusion

System virtualization on the emerging multi-core 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 multi-core servers favorably address. A typical multi-core server has the sufficient compute capacity for aggregating several of 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 applications. System virtualization addresses this problem since it enables the creation of virtual replicas of a system, over which different, independent VMs can be built, complete with their own, individual operating systems, software and applications. As a result, total software isolation of the VMs is possible and independent applications can be hosted in disparate virtual machines on a single physical server.

Apart from the software isolation, the key driver for virtualization adoption in datacenters
will be virtual machine performance and security 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 system virtualization architectures that provide CPU workload isolation. The number of CPU-cores in comparison to I/O interfaces is high in multi-core servers. As a result, different, independent virtual machines share the same I/O device. 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 architect-ed 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 on to a multi-core server, using virtual machines, performance interference caused due to shared resources across different 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.

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.

3. Device access path sharing, causes security vulnerabilities for all the VMs sharing the device.

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.

In recent years, there has been a spurt in interest to understand I/O virtualization architectures with a view towards, performance, security, isolation and scalability. Many initiatives [15, 44, 74, 41, 22, 46, 47, 81, 92, 77, 63, 77] have adopted various approaches and techniques to improve some or all aspects of the issues associated with I/O virtualization. Traditional OSes like Unix use a convenient but a highly complex mechanism for accessing I/O devices. All accesses to the I/O devices are necessarily routed through the privileged kernel. This is because the device is unaware of its sharing across multiple isolated processes. To preserve the process privacy from the device, the privileged kernel manages data transfers and movement to and from the device on behalf of any of the requesting processes. Virtualization compounds this complexity since most of the recent virtualization technologies adopt I/O device virtualization architectures preserving the existing I/O device access. The I/O operations impose performance penalties as high as five times over native device accesses [98, 41, 63, 53, 54]. Within the virtualization stack, many of the approaches have basically restricted themselves to either the hardware or software modes of virtualization and optimizations. Research groups that aim towards providing transparency and portability to applications seek to implement virtualization through hardware and associated hypervisor changes so that the Guest OS and application remain untouched. This makes the effort to move from legacy platforms to virtualized environments seamless. The most popular method for such a virtualization solution is by using device emulation. The biggest drawback in using device emulation is performance degradation, lack of security isolation and to certain extent scalability in sharing. The other approach is that of device para-virtualization wherein the Guest OS also participates in the virtualization stack. Of course, this involves changes to the Guest OS but the gain is in reduced performance overheads and improved security isolation. Nevertheless, this approach also suffers from scalability in sharing the device. However, both the predominant approaches fail to address the issue of resource usage control on the shared device, and leave it to the network stack to implement them. This approach results in a very coarse grained control on
resource usage and provides the loopholes for exploitation like denial of service attacks. With the advent of multi-core servers the problems of resource sharing can be offset, to some extent, by faster and higher bandwidth devices, but with the existing technologies the issues are not resolved, and also, they do not allow for good device utilization. Recent research publications [4, 97, 73, 44, 53, 56] have brought out the importance of enabling the I/O device for concurrent access. Again, concurrency can be provided on a single physical device through software [97, 46, 15, 4], or through hardware [44, 57, 15, 73]. Evaluation of different methods of concurrent I/O devices [98] gives an insight into the expected performance and security isolation benefits. From these publications clearly the future for I/O device virtualization lies in provisioning for safe and efficient concurrent access.

Performance, security and isolation have been the mainstream focus for a majority of the virtualization efforts. This thesis identifies two important issues that of scalability in sharing and provisioning for resource sharing QoS guarantees without compromising on performance and resource utilization. In any of the virtualized server usage environments, like the enterprise segment and the soft real-time application environment, these two properties will play a key role in adoption of the technology. It is no longer sufficient to ensure best-effort sharing, since on shared systems honoring SLAs will be the norm rather than an exception. The proposed I/O virtualization architecture in the thesis extends prevalent technologies and the PCI-SIG IOV specification. It does so by:

1. Using dynamic device reconfiguration properties on the NIC. This enables the creation of customized virtual NICs, that are driven by the resource usage QoS requirements of the VM. This provides fine-grained resource control and leads to a better device utilization without losing out on application performance.

2. Use of IOMMU features with vNIC to map its I/O space into VM’s address space. This not only ensures vNIC security isolation but also allows for direct access to the shared device from within the VM which provides fast and efficient device access. This leads to benefits of unchanged Guest OS ensuring legacy application hosting on the virtualized server.
3. Native device access through the vNIC ensures majority of resource usage resulting because of using the I/O device, accountable to the VM. This leads to better resource monitoring constructs, which are necessary to provide guarantees associated with resource SLAs. Native device access by the VM also improves security of the virtualization solution by restricting the device driver related failures to that VM only.

4. The vNIC device abstraction is defined on the hardware and hence relieves the hypervisor from managing I/O access on behalf of the VMs. This simplifies the hypervisor design and makes it more robust and reliable and also brings down the resource usage by the hypervisor.

Simulation and analysis of the LQN model of the proposed end-to-end I/O virtualization architecture demonstrates the following benefits:

- Significantly improved application performance.
- Improved scalability in its ability to share the device bandwidth across multiple VMs.
- Fine-grained resource usage specific QoS controls.
- Enabling simple but effective security to sharing of I/O devices in a virtualized server.

These benefits encourage to proceed with device designs that incorporate concurrency and enable QoS controls for better performance, utilization and security.

6.2 Lessons Learnt

I/O devices have been associated with indirect access mechanisms, and this has continued into the virtualization realm. Analyzing the end-to-end virtualization architecture has revealed the basic lacunae existing in the current approaches. Here in this section, we list below few of the crucial lessons learned in this exploration:

1. Virtualization enables transparent resource sharing improving their utilization. However, current practice of system design extends this to CPUs only. Other resources, specifically the I/O devices, are ill-designed for concurrent access.
2. Indirect access to I/O devices through privileged kernels has disadvantages of performance and security vulnerabilities. Safer, efficient and more robust systems can be built if the I/O devices are built for concurrency rather than implementing concurrency as part of the system software. This suggests that device management, and access should be separated.

3. Extending the singleOS systems to support virtualization is a weaker design choice since performance, safety and reliability are an after thought and tend to be built as wrappers on existing architecture. Instead, resource must be virtualization aware to allow native access from the Guest OS.

4. Basic goal of all virtualization efforts is to improve resource utilization. This demands the need for robust and fine-grained resource monitoring and control constructs. Enabling the device participation, in such controls, provides the necessary management without losing out on resource utilization.

By virtue of studying and validating I/O virtualization architectures we conclude that designing systems from an end-to-end perspective enables greater flexibility in managing resources for virtualization and delivering additional benefits of performance and security. We observe that both the characteristics, performance and security, can be handled with simple, elegant constructs that are built on hardware APIs.

6.3 Future Work

The end-to-end virtualization architecture analysis brings up many interesting challenges that give an opportunity for innovation. Here we list some of the ideas that need further exploration:

1. The proposed architecture demands revisiting NIC design to provision for the required properties. It would be interesting to build a physical prototype for the same and implement the end-to-end virtualization architecture to evaluate the actual system.

2. The idea of separating device access from management and enabling the storage device to be virtualization aware is another aspect that could lead to some interesting results.
Large storage solutions are now a judicious mix of both disk and networking devices. How to make the devices virtualization aware to meet resource provisioning goals is an open question.

3. Hypervisors can be a single point of failure for virtualized servers. Building robust and trust-worthy hypervisors becomes mandatory to ensure reliability in virtualized servers. Building hypervisors using micro-kernel architectures to decouple the resource management from access is another exciting idea for exploration.

4. Using reconfigurable processors within the NIC is yet another interesting area one could explore. Although the thesis suggests reconfiguration to be used in device memory allocation, it can be extended to implement adaptive allocation policies. Device reconfiguration can also be used to associate specific ASICs for encryption methods to a vNIC.

5. The thesis opens up an interesting area for exploration that of re-assessing system design and architectures to enable virtualization mechanisms that guarantee performance without sacrificing security or resource utilization. One exciting possibility is that of using reconfigurable system design to enable virtual resources that can be built using hardware APIs. As demonstrated by this thesis, extending hardware design to support performance and security isolation provides a means for robust and elegant system architectures.

6. In the thesis, we have used LQNs as a tool for evaluating the architecture. This is because the basic aim was to evaluate the architecture for improvement. LQNs gives a simple, easy to use method for analyzing the proposed architecture for its feasibility and meeting of the design goals. To this extent, the LQNs package developed by RADs lab of Carleton University was sufficient to develop and analyze the required model of the virtualization architecture. However, a number of assumptions on the arrival and service time distributions have been made to make the analysis of the queuing model. It would be worthwhile to extend the software to incorporate different distributions to model the arrival rate and service time. Furthermore, it would be interesting to extend the model to
incorporate the idea of workload adaptivity to improve resource utilization without compromising on application performance. This would immensely benefit the consolidation efforts using virtualization as the means. Any application workload, that has a significant component of I/O in it, requires both the I/O device bandwidth as well as associated CPU processing to effectively deliver application performance. This work has shown that decoupling the hypervisor, to just manage resources rather than participate in data access, makes the virtualization architecture robust, secure and efficient. An interesting extension to this thought is incorporating adaptivity, based on the expected workload to resource provisioning. This is a suggested future work possibility.

6.4 Summary

In this chapter, a gist of the thesis is given. The motivation for the work with a clear listing of the observed lacunae in the current design and methodology of I/O device virtualization is highlighted. Which, leads to the specification of the design goals, and development of device and end-to-end I/O virtualization architecture to meet the stated goals. Further, description of evaluation method and its suitability for the purpose is explained and observed benefits are enumerated. The chapter concludes with a listing on lessons learned and some interesting possibilities for future work.
Bibliography


[37] Franks Greg and et Al. Layered queueing network solver software package, ”rads”


