VIRTUALIZED RESOURCE MANAGEMENT IN HIGH PERFORMANCE FABRIC CLUSTERS

A Thesis
Presented to
The Academic Faculty

by

Adit Ranadive

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
December 2015

VIRTUALIZED RESOURCE MANAGEMENT IN HIGH PERFORMANCE FABRIC CLUSTERS

To my parents who have strived hard to give me so much,

To my brother, Rohan, to help me get started and

To my wife, Smita for her immense love and support

ACKNOWLEDGEMENTS

I would like to begin by thanking my parents who have sacrificed much to give me the opportunity to come to the US to pursue graduate studies. Without their encouragement and guidance this thesis would not have been possible. They have supported me through all these years with the ups and downs that exist in this PhD life. They supported my decision to go through the PhD program even though it would have been easier to accept that full-time job after MS so I could have been ‘settled’ in life. Without my father’s advice I would not be the person that I am today.

I would like to express my heartfelt gratitude to my advisors, Prof. Karsten Schwan and Dr. Ada Gavrilovska. They have shaped my approach to systems research immensely on how to approach the problem and what is the best course for the solution. They have been very patient with me as I sometimes struggled to distinguish between the research contributions and the exact implementation details. They were very understanding when I wanted to accept the full-time opportunity at VMware so that I could start after finishing the proposal. I owe a lot to Ada as well for helping me work through several iterations of papers, results and taking those phone calls to help me help her understand what I was actually trying to achieve. I will sorely miss Karsten in this new phase of life.

I owe an immense thanks and love to my wife, Smita whose unwavering support through my life as a graduate student from MS to PhD has got me through the worst of days. Her constant supply of encouragement, home-cooked food and chai was what I would look forward on several days when we started living together. Even though we stayed apart for four years before she moved to the US, her words would help me stay motivated several times.

I would also like to immensely thank my brother, Rohan and his wife Wendy, for helping me settle in a new country and letting me crash in their house whenever the long weekend beckoned to let me leave my graduate life worries behind me. Without my brother's support and guidance I would have found life much harder than it was.

I thank my proposal and defense committee members, Prof. Sudhakar Yalamanchili, Prof. Nick Feamster, Prof. Ellen Zegura, Prof. Ling Liu and Prof. Doug Blough for their valuable insights and helping shape the thesis to what it is. I would like to thank Chad Huneycutt for helping me setup the cluster to conduct my research on. There have been several times where I would turn to him to fix an issue which I could not diagnose remotely. My thanks go out to Susie McClain for all the support she has provided to us students. To my current manager at VMware, Shelley Gong, I would like express thanks for her patience and understanding while I was finishing my Ph.D.

Finally, there have been so many labmates, friends, now colleagues as well, without whose support and encouragement I would not have been able to finish this thesis: Mukil Kesavan, Anand Zaveri, Priyanka Tembey, Vishakha Gupta, Romain Cledat, Tushar Kumar, Dulloor Rao, Jay Lofstead, Himanshu Raj, Danesh Irani, Vijay Arvind, Karishma Babu, Bhuvan Bamba, Neeti Wagle, Rakshita Agrawal, Raghav Vijaywargiya, Nawaf Almoosa, and many others. Thank you all for your kind words and bearing with my eccentricities.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Managing virtual resources in existing cloud computing infrastructure is critical to ensure that applications deployed are provided with the required isolation and performance guarantees. With high performance RDMA-based fabrics with capabilities like VMM-bypass, SR-IOV becoming ubiquitous in the datacenter, the workloads deployed in these VMs are becoming more latency-sensitive as well as bandwidth-intensive to utilize the network. Thus, as applications are collocated on such networks, the sharing of these networks can lead to significant performance degradation for the applications.

In order to manage this performance degradation and hence the network resources allocated to the VMs, there exists several constraints that this thesis aims to address. First, the hypervisor does not have any direct insight to usage of the network by the VMs since the data transfer bypass the hypervisor completely. Second, without any monitoring information the hypervisor cannot make fine-grained resource allocations in order to provide isolation and performance guarantees which is required for such workloads. Third, integrating these networks into current software-defined fabric management solutions would not be possible without first having these mechanisms.

This thesis addresses these constraints as follows: (1) we develop IBMon, a hypervisor-based tool to help monitor such fabrics through memory introspection to find the current network conditions experienced in terms of bandwidth or latency, and is the basis for our resource management mechanisms, (2) we observe the application characteristics with which interference occurs and use them to build a resource management system called FaReS, for collocated data-intensive workloads, (3) we develop Economics-based resource management mechanisms to limit and provide isolation and performance guarantees for latency-sensitive applications on a per-node (Resource Exchange) or multi-node (Distributed Resource Exchange) basis (4) we develop our fabric management solution, Sphinx that leverages our Economics-based mechanisms to provide latency proportionality to collocated latency-sensitive and data-intensive applications.

We use the mechanisms described in this thesis to manage both Paravirtual as well as SR-IOV-based InfiniBand clusters running Xen and KVM hypervisors respectively. We also utilize the Floodlight OpenFlow controller with Sphinx to orchestrate the SR-IOV InfiniBand network to provide latency guarantees for consolidated workloads. The applications deployed include low-latency financial codes, distributed data-intensive applications and distributed MPI workloads to highlight the utility and effectiveness of our resource management techniques to provide application-level guarantees and performance isolation.

CHAPTER I

INTRODUCTION

Workloads such as multi-tier enterprise codes [8], latency-sensitive applications like financial trading [126, 45], VoIP services [74], in-memory databases [59, 33, 137] and distributed transaction workloads [136] exhibit communication-intensive and latency-sensitive properties that rely on optimized networks for best performance. These high-performance networks, or 'high performance fabrics', enable the best possible performance to applications via mechanisms like CPU-offload, OS-bypass and Zero-Copy capabilities that minimize buffering overhead, protocol processing costs and loss.

Modern virtualization and cloud platforms support the use of high performance fabrics (through software [78, 109] and hardware-based [58] methods), and thus permit these applications to be easily deployed. However, current solutions do not support the necessary resource provisioning required to dynamically manage resource allocations, and therefore limit the ability to consolidate these types of applications and achieve sought-after virtualization benefits like increased platform utilization. This is because the techniques used by such fabrics to achieve high performance allows the VM-interconnect interactions to completely 'bypass' the system software, thus, *preventing any fine-grained I/O management that is critical to provide performance isolation, fairness and guarantees in such shared virtualized platforms.* As a result, such workloads continue to be run in isolated, non-virtualized platforms, leading to server over-provisioning and lowered utilization.

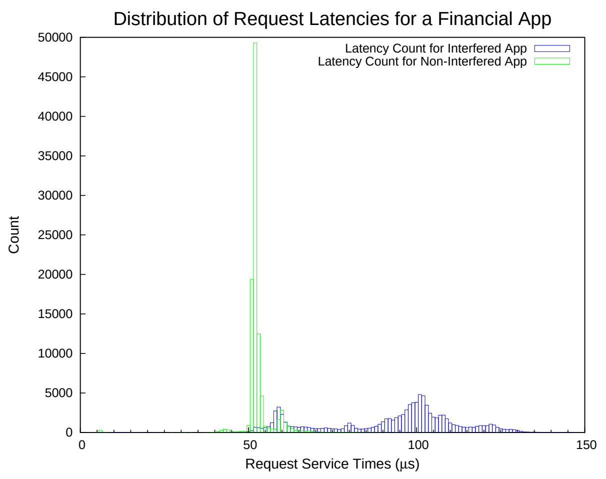


Figure 1: Distribution of Latencies for a Latency-Sensitive Application collocated with a Data-Intensive one.

1.1 Motivation

We illustrate this key problem in Figure 1, which shows the impact on latency distribution of a latency-sensitive workload when collocated with a data-intensive workload on a modern virtualized platform with support for a high performance fabric. We observe that with consolidation, the latency and latency variability of the latency-sensitive workload degrade, despite hardware-level support for isolation. This highlights that the platform is unable to provide any performance isolation or reduce I/O interference between these applications. Current state of the art approaches [26, 39, 40, 56] that offer performance isolation for networks cannot be applied to such fabrics since they use software rate-limits that performs data buffering within the OS.

Another aspect of these communication-intensive workloads concerns their deployment across *multiple nodes and VMs as distributed VM Ensembles (VMEs).* This implies that there exist a number of VMs pertaining to a single application communicating across the cluster which can lead to varying degrees of I/O interference across multiple hosts. This further complicates how I/O resource provisioning must be performed, which must consider the cluster-wide impact of the resource management actions as well. While current efforts [14, 122, 42] have considered the distributed impact for some of these workloads, they cannot be directly applied to a virtualized platform with high performance fabrics for the reasons mentioned above.

Furthermore, as network management gets decoupled from the hardware itself, network resource provisioning should become ubiquitous through abstractions that enable such performance isolation, fairness and guarantee properties for these types of applications. Current cloud environments that leverage Software-Defined Networking (SDN) [101, 88], however, provide a fairly static isolation model and are incompatible with current high performance fabrics. This prevents such platforms from providing the necessary fine-grained I/O management required to satisfy these properties via various user-defined policies.

When such diverse applications are collocated on shared virtualized platforms there are many factors that contribute to the overall 'I/O interference' or 'performance degradation' experienced by latency-sensitive and data-intensive applications: (i) type of the virtualization platform used can impact the isolation and degree of consolidation provided to these workloads, (ii) the I/O characteristics of these applications would determine the degree of isolation as well as guarantees that can be provided, (iii) the communication matrix of such distributed applications can identify hotspots or interference links in such clusters.

These factors help us identify the following management constraints for current shared virtualized platforms (i) without the hypervisor's ability to monitor the workload's I/O characteristics or VMs usage of these fabrics, the platform cannot provide isolation and performance guarantees, (ii) since the hypervisor cannot provide fine-grained I/O provisioning or perform management decisions for VMs due to the techniques used to achieve high performance, this reduces the degree of consolidation that can be supported on these platforms, (iii) without such support it is harder to integrate these fabrics into emerging cloud computing platforms and datacenter fabric management solutions, and (iv) these constraints are particularly challenging for workloads spanning multiple VMs, utilizing physical resources distributed on multiple nodes and interconnect fabric.

1.2 Thesis Statement

Novel I/O provisioning methods within existing system and network management software can provide performance isolation, fairness and guarantees to communication-intensive and latency-sensitive workloads deployed in virtualized high performance fabric clusters.

1.3 Approach and Contributions

We support the thesis statement through our general management model and our contributions which we describe next.

1.3.1 General Model

This thesis uses an 'online and dynamic' model to help maintain the performance fairness and guarantee requirements specified through several user-defined policies. In case of fairness, these policies identify how the oversubscribed network I/O resource be shared between the collocated applications, while for guarantees they recommend an 'SLA value' that needs to be satisfied. Since the VM data bypasses the hypervisor, the online model first finds the network I/O usage for a VM through the indirect, asynchronous method of reading memory pages shared with the device during the runtime of the application. This identifies whether the I/O resource is split in the fashion as dictated by the policy. If not, the model then has to find the VMs (culprit VMs) that are receiving more than their required split or are causing interference in case of guarantees. The model then must adjust resources allocated to the culprit VMs in order to reduce the I/O they are performing. The resources allocated also depend on the impact the new allocation will have on their I/O usage in the future. Such an approach can lead to dynamism which is handled by the model by 'continuously' ensuring that the fairness and guarantee criteria are maintained during the lifetime of the workload.

Our model is capable of supporting multiple policy definitions through its dynamic mechanism. Policies concerning fairness can be supported in two ways: (i) through specification of how the I/O resource should be ‘split’ between competing applications, thus enabling *Proportional Resource* consumption, (ii) through specification of relative performance degradation which enables *Distribution of Degradation* between collocated applications. We also support policies that specify the application performance guarantee that needs to be maintained.

In order to support such diverse policies for collocated communication-intensive and latency-sensitive workloads, our management model must interpret the impact of allocation of resources on the I/O objective as specified by the policy. Essentially, our online component gauges the impact of resource allocation to actual resource usage as well as impact on application performance, i.e. finding the ‘Resource Sensitivity’ for the application. The resource sensitivity information can be measured in terms of rate of change of I/O usage for an application for example.

The resource sensitivity values will be different for these applications since their I/O characteristics are different. For example, latency-sensitive applications can be ‘more’ sensitive to resource changes while communication-intensive applications might be ‘less’ sensitive. To incorporate these differences between various applications as well as the difficulty of fine-grained allocation and monitoring we implement a set of abstractions and specific model instances that ‘normalize’ the resource allocation provided to applications in order to meet the different policy objectives. We describe these within as our contributions next.

1.3.2 Contributions

(i) IBMon – a lightweight monitoring tool to find the I/O characteristics of applications that utilize such high performance fabrics. It leverages knowledge about application-device interactions and how such information is stored in the VM's memory. The information is extracted by using hypervisor-level memory introspection techniques to read the relevant memory of the VM and provides insights into the VM I/O behavior, including communicating endpoints, request patterns and data volume. This allows us to infer the usage of the underlying network by the VM without any perturbation to the application.

This monitoring information can be provided through hypervisor extensions and used to enable resource management decisions that were not possible before for such high performance fabrics. IBMon allows us to perform 'timely detection' of I/O characteristics from which we can infer possible I/O interference for these applications. This allows us to provide monitoring information for both latency-sensitive and throughput-intensive applications in order to support the diverse policies in our management model. IBMon is implemented for both Xen [16] and KVM [67] hypervisors and the InfiniBand interconnect, is used as an example of a high performance fabric throughout this thesis.

(ii) FaReS (Fair Resource Scheduling) – a resource management model which uses monitoring data from IBMon and controlled CPU allocations to provide fairness guarantees to data-intensive VMs using VMM-bypass on fabrics like InfiniBand. FaReS relies on online observations made for application performance as compute and I/O characteristics for collocated applications are changed. These observations, in conjunction with the monitoring data from IBMon, are used to drive CPU Capping policies in order to provide 'fairness guarantees' to the VMs.

We model these guarantees on 'Proportional Resource' consumption or conversely 'Performance Degradation' allocated to a VM. FaReS deducts CPU cycles from VMs based on the overage of data sent by the VM. Since the OS is not involved in data transfer we reason that this constitutes a penalty for the VM in order to reduce the I/O performed. For various combinations of High Performance Computing or communication-intensive workloads we show that FaReS can maintain various levels of fairness.

(iii) ResEx (ResourceExchange) – a cluster-wide resource management model based on microeconomics that utilizes IBMon to provide applications certain performance guarantees. It introduces our 'Virtual Currency Abstraction' that allows us to dynamically map the resource sensitivity value of latency-sensitive applications to the underlying physical I/O resources. This uses microeconomic principles like Congestion Pricing to coordinate the use of the shared interconnect on a per-host basis, and to provide performance guarantees to latency-sensitive applications.

The evaluation of these mechanisms uses *Nectere*, a latency-sensitive benchmark developed in our work, that is modeled on financial applications. Nectere can be configured which specific CPU and I/O characteristics in order to evaluate behaviors of such applications on virtualized high performance fabric clusters.

The flexibility of ResEx is demonstrated through implementation and evaluation of two different policies with different goals based on Congestion Pricing - FreeMarket (maximize resource utilization) and IOShares (lower latency variation). We use our observations on I/O latency behavior to meet the latency guarantees specified by these policies by dynamically adjusting the interfering VM's I/O usage through CPU allocations. These results demonstrate that latency guarantees can be provided to collocated applications on such platforms through novel effective monitoring and management mechanisms.

(iv) DRX (Distributed Resource Exchange) – is an extension to our resource exchange model to consider distributed applications and reducing their impact on latency-sensitive applications deployed on hardware-virtualized (SR-IOV) InfiniBand cluster by enabling resource management policies at a cluster-level.

DRX identifies distributed applications and its components via VM Ensembles or VMEs and interactions between VMEs and their VMs are modeled on a transitivity-based property called the Distributed Causal Congestion Relationship (DCCR). This relationship lets us identify the culprit(s) that impact the latency-sensitive application collocated with the distributed applications. By utilizing Congestion Pricing on a per VME-basis we can divide cluster-wide network resources through various policies in order to improve performance of latency-sensitive codes.

We evaluate DRX with a combination of data-intensive (Hadoop and High Performance Linpack) and latency-sensitive (Nectere) benchmarks for various deployment scenarios. These results show that by factoring in the 'distributed interference' caused by data-intensive applications to the resource provisioning methods allows latency-sensitive applications to achieve their performance guarantees when collocated on such fabrics.

(v) Sphinx – is an implementation of a software-defined network management platform that builds upon the above mechanisms and provides *Latency Proportionality (LP)* – a new fairness property – to collocated latency-sensitive applications through dynamic orchestration of network resource allocations.

Latency Proportionality identifies the performance degradation that can be experienced by collocated workloads, depending on user-defined policies. Sphinx utilizes a Congestion Pricing-based controller to dynamically maintain LP for collocated latency-sensitive applications.

Sphinx is implemented on top of the OpenFlow Floodlight Controller and manages a SR-IOV InfiniBand cluster by utilizing the underlying hardware mechanisms for apportioning I/O for VMs. We demonstrate that Sphinx can dynamically achieve LP for various applications I/O characteristics and user-defined policies.

1.4 Thesis Organization

The rest of the dissertation is dedicated to quantitatively proving the thesis statement and is organized as follows. We provide a background on capabilities afforded by high performance fabrics and provide key details for InfiniBand, an example fabric in Chapter 2. Chapter 3 describes our InfiniBand monitoring extensions and how an existing hypervisor can be enlightened to provide VM's network usage of high performance fabrics.

Chapter 4 discusses FaReS, our research contribution for leveraging an enlightened hypervisor to make fairness-based decisions for scheduling HPC applications. We describe our research contribution ResEx that uses a Virtual Currency Abstraction for resource management to improve latencies for low-latency workloads in Chapter 5. This chapter also presents our configurable latency-sensitive benchmark, Nectere, which we use to evaluate our platforms.

Chapter 6 presents our Distributed Resource Exchange model that considers distributed applications when performing resource allocation decisions.

Chapter 7 discusses how we can extend existing software-defined platforms to orchestrate the network to provide Latency Proportionality to collocated applications.

Chapter 8 discusses current and past research related to this dissertation. We provide a summary of the thesis in Chapter 9 and list possible future directions in Chapter 10.

CHAPTER II

BACKGROUND ON HIGH PERFORMANCE FABRICS

We motivated the need to provide monitoring and management methods for high performance fabrics in the previous chapter. However, we did not delve into the architectural and design details of such fabrics that motivate our novel resource provisioning methods. In this chapter, we expand on these details and highlight how these fabrics are deployed in virtualized environments.

2.1 Capabilities of High Performance Fabrics

We mentioned three capabilities in the previous chapter that identify networks as ‘high performance’. CPU-offload, OS-bypass and Zero-Copy are features that identify these networks. CPU-Offload permits protocol stack processing on the NIC hardware rather than in software. OS-bypass [133] allows the application to directly communicate with the NIC hardware through memory-mapped I/O pages to issue network requests. Zero-Copy occurs as a result of OS-bypass and programming DMA hardware to read/write safely and directly to application buffers without the need of the OS to verify the operation.

In general, these capabilities together are referred to as providing RDMA or Remote Direct Memory Access and network hardware is referred to as a Host Channel Adapter or HCA. In order to take advantage of the RDMA characteristics, applications cannot use the standard sockets API for communication and must utilize another programming model called the ‘Verbs API’ [52]. We illustrate the differences between these two APIs in Figure 2, where data in the sockets-based API traverses the entire network stack while in case of RDMA, the network hardware can directly read the data from the application due to the OS/VMM-bypass functionality.

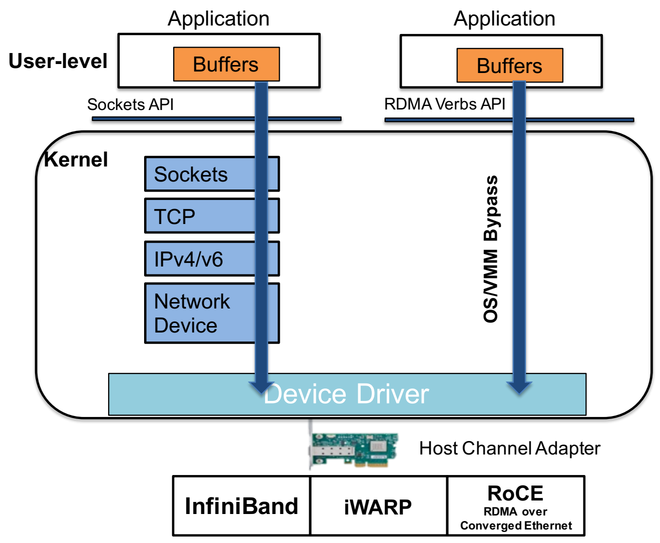


Figure 2: Comparing Sockets and RDMA APIs for communication.

Examples of such fabrics include InfiniBand [52], Myrinet [90], Quadrics [107] and more recently RoCE (RDMA over Converged Ethernet) [54] and iWARP [28]. Using these capabilities these fabrics can provide extremely low latencies ($< 5\mu s$) and high bandwidths (>40Gbps). In this thesis, we use InfiniBand as an example high performance and RDMA-capable fabric. We next highlight the architectural details of IB and their usage models in virtualized environments.

2.2 InfiniBand Communication Model

The communication model used in IB is based on the concept of queue pairs (Send and Receive Queues). A request to send/receive data is formatted into a Work Request (WR) which is posted to the respective queues. There are also Completion Queues (CQs) in which Completion Queue Entries (CQEs) containing information for completed requests are stored. Whenever a descriptor is posted to the queue pair (QP), certain bits called 'Doorbells' are set in User Access Region (UARs) buffers. The UARs are 4KB I/O pages mapped into the process' address space. These pages are shared directly between the application and the HCA and permits the OS-bypass functionality. When a request is issued, the doorbell is "rung" in the process' UAR.

The HCA will then pick up these requests and process them.

2.3 InfiniBand Memory Management

The HCA maintains a table called the Translation and Protection Table (TPT). This table contains the mappings from physical to virtual addresses of the buffers and queues described above. For InfiniBand, the buffers need to be registered with the HCA, and these registry entries are stored in the TPT. Also, these buffers must be kept pinned in the memory during data transfer so that the HCA can DMA the data directly in and out of host memory, thus enabling Zero-copy. Each entry in the TPT is indexed with a key that is generated during the registration process.

2.4 Paravirtual or VMM-bypass InfiniBand Driver Model

When virtualized in Xen, the device driver model is based on the ‘split device driver’ model as described in [16]. It consists of a Backend and Frontend Driver. The frontend driver resides in the guest OS kernel while the backend driver resides in the dom0 kernel. In the case of regular devices, all requests to access the device from the guest VM must travel from the frontend driver to the backend driver. Therefore, both the data and control path instructions are sent to Dom0.

For InfiniBand, the split-driver model is used slight differently. The control path operations from the guest VM, like memory registration and queue pair creation, must all go through the backend driver. Fast path operations like polling the CQ and QP accesses are optimized using VMM-bypass [78]. The data path is highly optimized since the HCA can write DMA data directly to the target buffer. As a result, IB platforms can be virtualized with negligible impact on the attainable latency and bandwidth.

2.5 Hardware-virtualized InfiniBand Model

In this section we provide a background on hardware-virtualization and how they are used in conjunction with IB network hardware.

2.5.1 Single Root I/O Virtualization (SR-IOV)

With SR-IOV [32], the physical device interface (i.e., the device Physical Function (PF)) and associated resources are 'partitioned' and exposed as Virtual Functions (VFs). One or more VFs are then allocated to guest VMs in a manner that leverages PCI passthrough functionality.

2.5.2 PCI Passthrough

PCI Passthrough allows PCI devices (SR-IOV-capable or standard) to be directly accessible from within the VMs, without the involvement of the hypervisor or host OS, but requiring Intel's VT-d [2] or AMD's IOMMU [9] extensions for correct address translation from guest physical addresses to machine physical addresses [18]. The hypervisor (e.g., KVM or Xen) is responsible for assigning the PCI device (specified for passthrough) to the guest's PCI bus and removing it from the management domain's PCI bus list – i.e., the device is under full control of the guest domain. While providing guests with near-native virtualized I/O performance, by bypassing the management domain, i.e., the hypervisor, it becomes challenging to monitor and manage the guest's I/O behavior.

2.5.3 SR-IOV InfiniBand

We utilize Mellanox ConnectX-2 InfiniBand devices in our work which provide SR-IOV support by dividing the available physical resources, i.e., queue pairs (QPs), completion queues (CQs), memory regions (MRs), etc., among VFs and exposing this subset of resources as a VF. The PF driver running in the management domain is responsible for creating the number of VFs (in our case 16). Each of these VFs are assigned to a guest using PCI passthrough. A Mellanox VF driver residing in each guest is responsible for device configuration and management.

CHAPTER III

IBMON: MONITORING VMM-BYPASS CAPABLE INFINIBAND DEVICES USING MEMORY INTROSPECTION

In the previous chapter, we described the capabilities afforded by high performance fabrics and the architectural details that provide high performance. We utilize those details to design our monitoring tool, IBMon for such fabrics which we describe in this chapter. We begin by outlining the need for IBMon-like tools in providing better resource management in virtual environments. We describe the design details of IBMon using memory introspection for two different virtual environments and show how IBMon can be used to provide accurate bandwidth information for RDMA-based applications.

3.1 Importance of monitoring VM resources

Virtualization technologies like VMWare [131] and Xen [16] have now become a de-facto standard in the IT industry. A main focus of these technologies has been to manage the virtual machines (VMs) run by the virtual machine monitor (VMM) or hypervisor, in order to ensure the appropriate allocation of platform resources to each VM, as well as to provide for isolation across multiple VMs. Such actions are necessitated by the fact that static resource allocation policies are not typically suitable, as they imply apriori knowledge about the VM's behavior – which may be impossible to determine in advance, and/or worst-case analysis of its resource requirements – which may lead to significant reductions in the attained resource utilization and platform capacity.

Managing virtualized platforms relies on mechanisms that dynamically adjust resource allocations, based on workload characteristics and operating conditions, as well as on current VM behavior and resource requirements [65, 127, 139]. Such techniques are particularly relevant to distributed clouds and virtualized data centers, to more effectively utilize aggregate platform resource and reduce operating costs, including for power. The latter is also one reason why virtualization is becoming important for high end systems like those used by HPC applications.

A basic requirement for effective resource management is the availability of runtime information regarding VMs' utilization of platform resources, including CPU, memory, and network and disk devices. Using this information, the VMM infers the behavior and requirements of VMs and makes appropriate adjustments in their resource allocations (i.e., it may increase or decrease the amount of resources allocated to a VM) [65, 127], while still ensuring isolation and baseline performance levels. Monitoring information is collected continuously in the hypervisor and/or its management domain (i.e., dom0 in Xen). For instance, for scheduling CPU resources, the VMM scheduler relies on monitoring the time a VM has spent executing or idling in order to adjust the corresponding scheduling priorities or credits [140], whereas for memory it monitors the VM's memory footprint, and if necessary, makes appropriate adjustments, e.g., by using the 'balloon-driver' in the case of the Xen VMM. The same is true for most devices, for which the VM's device accesses are 'trapped' and redirected to a centralized device management domain, i.e., following the 'split device driver model' in Xen [16, 124]. All information regarding the VMs' utilization of a particular device therefore, is centrally gathered and can be easily maintained and used to enforce limits and/or orderings on device accesses.

Furthermore, management actions may span multiple types of resources, such as when monitoring information regarding the network buffer queues is used to adjust the CPU credits for a VM, so as to eliminate buffer overflow and ensure timely response [65, 127]. Stated more technically, the management of the virtualized platform relies on the ability of the virtualization layer (e.g., ESX server or the Xen hypervisor plus dom0) to have the ability to monitor and control the utilization of each of the platform resources on behalf of a VM.

Unfortunately, some IO devices do not follow the 'split driver' model, and cannot be easily monitored via the same approach as above. Some such devices exhibit 'self-virtualization' capabilities [48], i.e., the management logic that controls which VM will be serviced by the device is offloaded onto the device itself. When used in the high performance domain, in order to meet the high bandwidth/low latency messaging requirements of HPC applications, these devices also use their RDMA capabilities to read/write data directly from/to guest VMs' address spaces. Specifically, the control path invoked during a VM's setup and device initialization is still routed through a management domain such as dom0, but subsequent device accesses are performed directly, thereby 'bypassing' the VMM. Devices virtualized in this manner include InfiniBand HCAs [51, 78], and they will also include next generation high performance NICs such as those used for high end virtualized IO [48, 108] and as facilitated by next generation device and board interconnection technology [28, 129].

The challenge addressed by our research is the acquisition of monitoring information for devices that use VMM bypass. The outcome is a novel monitoring utility for InfiniBand adapters, termed *IBMon*. IBMon's software-based monitoring methods can provide the information they collect to the management methods that require it. An example is the provision of information about communication patterns that is then used to adjust a VM's CPU or memory resources, or to trigger VM migration. IBMon can also supplement an IB device's monitoring logic, or it can compensate for issues that arise if device-level monitoring is not performed at the levels of granularity required by management. Most importantly, by using VM introspection techniques [38, 13], IBMon can acquire monitoring information for bypass IB devices, at small overheads and with minimal perturbation. Specifically, for such devices, IBMon can detect ongoing VM-device interactions and their properties (i.e., data sent/received, bandwidth used, or other properties of data transfers). Such determination of the VM's usage of its share of IB resources is critical to proper online VM management. Using introspection, of course, also implies that IBMon's implementation uses knowledge about internals of the IB stack and buffer layout in the VM's address space. Stated differently, due to the absence of hardware-supported device monitoring information, IBMon uses gray-box monitoring approaches to assess the VM-device interactions and their properties.

The current implementation of IBMon enable asynchronous monitoring of the VM's bandwidth utilization, however, as noted earlier, the same approach can be used to monitor other aspects of VMs' usage of IB resources. While IBMon is implemented for InfiniBand devices virtualized with the Xen hypervisor, its basic concepts and methods generalize to other devices and virtualization infrastructures. Of particular importance are the novel ways in which the virtualization layer asynchronously monitors VM-device interactions that bypass the VMM (i.e., for high performance), and the insights provided regarding requirements for hardware supported monitoring functionality.

In the remainder of this chapter, we first describe IBMon and evaluate its utility and costs. In the following sections, we then provide brief summaries of the techniques on which IBMon depends and next present its design and implementation. Section 3.5 describes the bandwidth estimation mechanism currently used in IBMon. We next show experimental results to show that with IBMon, we can asynchronously monitor VMM-bypass operations with acceptable accuracy and negligible overheads, including as we increase the number of VMs.

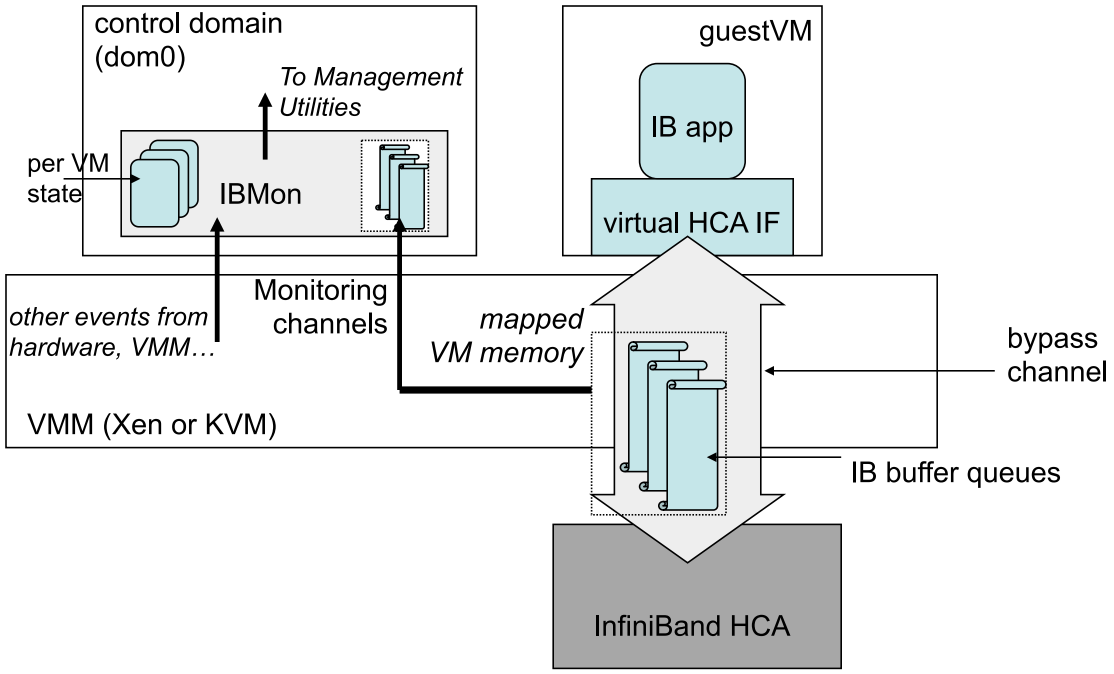


Figure 3: IBMon Utility

3.2 Virtual Memory Introspection

The design of the IBMon tool for the Xen hypervisor is specific to (1) the manner in which IB devices export their interfaces to applications (and VMs) and the memory management mechanisms they support, (2) the manner in which these devices are virtualized in Xen environments, and (3) the memory introspection techniques being used.

We have previously described the InfiniBand architecture and communication model related to (1), (2) in Chapter 2 and describe the relevant information for (3) in this section.

With the Xen VMM, pages of one VM can be read by another VM simply by mapping the page into the target VM memory. This concept was first introduced by Garfinkel and Rosenblum [38]. Xen contains the XenControl library that allows accessing another VM's memory. Using the function *xc_map_foreign_range*, the target memory is mapped into the current application's virtual memory. We use this mechanism to map the physical pages which correspond to the IB buffers into the monitoring utility.

3.3 Design

Figure 3 illustrates the design of the IBMon utility. It is deployed in the VMM's control domain, i.e., dom0 in Xen, and its goal is to asynchronously monitor the per VM utilization of the resources of the InfiniBand fabric, i.e., bandwidth utilization in our current realization.

As described above, the IB HCA does have the capability to export virtual interfaces, i.e., queue pairs and completion queues, directly to guest VMs. However, the hardware only exports counters about the aggregate usage of the communication resources used by all VMs, which is not adequate for the per-VM management decisions we would like to support with IBMon. Therefore, we rely on knowledge about the layout of the memory region used by each VM for its virtual HCA interface to map and monitor the queues residing across those memory pages. We then apply memory introspection techniques to detect changes in the page contents, and based on that make conclusions regarding the VMs use of IB bandwidth.

The monitoring process is triggered periodically, with a dynamically configurable frequency. The exact details of how the current implementation uses the accessed data to perform bandwidth estimations is described later in Section 3.5. Dynamic changes in monitoring frequency are necessary to adjust the sampling rate to the frequency at which events being monitored are actually occurring, so as to avoid degradation in the quality of the monitoring information.

Note that the in the case of IB, the HCAs do not maintain per work queue bandwidth or timing information in hardware. This, coupled with the asynchronous nature of both the monitoring operations as well as the VM's IO operations themselves, requires that we adopt the dynamic approach described above. The presence of such hardware supported information in next generation IB or other multi-queue, VMM-bypass capable devices will simplify the processing required by IBMon or similar monitoring utilities, but it will not eliminate the need for such information to be asynchronously accessed and then integrated with the platform level management mechanisms.

In addition, while the presence of adequate hardware support may obviate the need for the bandwidth estimation mechanisms integrated in IBMon, there will be other types of information regarding the VMs' device accesses which will still require that we integrate memory introspection mechanisms with the monitoring tool. Examples of such information include certain patterns in communication behavior, identification of the peers with which the VM communicates with (e.g., to determine that it should be migrated to a particular node), IO operations involving certain memory region (e.g., to indicate anomalous behavior or security issues), etc.

Finally, the figure illustrates that IBMon can interface with other management or monitoring components in the system. This feature is part of our future work, and the intent is to leverage the tool to support better resource allocation (i.e., scheduling) policies in the VMM (e.g., by using the IBMon output to make adjustments in the credits allocated to VMs by the VMM scheduler), or to couple the monitoring information with large-scale 'cloud' management utilities.

3.4 Implementation

IBMon is implemented as a monitoring application which runs inside dom0 in Xen. It uses the XenControl library (libxc) to map the part of the VMs' address space used by the IB buffers. In the InfiniBand driver model, upon the front-end driver's request, the back-end driver allocates the buffers and then registers them with the HCA with the help of the dom0 HCA driver. IBMon augments the back-end operations to establish a mapping between the physical memory addresses and the types of buffers they are used for (CQ, QP, etc.), for each VM. The table is indexed by the VM's identifier and is accessed through the '/proc' interface provided by the back-end.

The IBMon utility reads the entries of the table when the guest VM starts running an IB application. It detects any new entries in the backend driver table. For each of the entries, the utility uses the xc_map_foreign_range function of libxc to map that IB buffer page to its address space. Once the page is mapped, the VM's usage of the IB device can be monitored. Note that these pages are mapped as read-only to IBMon's address space.

3.5 Estimating VMs' Bandwidth Utilization with IBMon

InfiniBand uses an asynchronous (polling-based) model to infer completion of communication. Hence, any IB-based application must poll the CQ to check whether a request has completed or not. These CQ buffers are also mapped to IBMon's memory space, which itself can then check and determine whether any work requests on behalf of a specific VM have completed. The CQ buffer contains several fields which can be used for bandwidth estimation. First is the number of completion entries (CQE) which indicates how many completion events have occurred. Second is the byte length (byte_len) field which indicates how many bytes were transferred for each of the completion event. These entries are written to directly by the HCA.

IBMon periodically inspects the newly posted CQEs, gathers the amounts of bytes transmitted during the latest time interval, and uses that to estimate the bandwidth. There are several challenges in this approach. First, the CQEs do not contain any timing information. Therefore, if IBMon is configured with a short monitoring interval, IBMon detects CQE which indicates that a work request for a read of a large buffer just completed, using a simple Bytes/time formula it will compute bandwidths which far exceed the fabric limits. Since we do not know when that read operation was initiated, we cannot just adopt a model where we compute the bandwidth by averaging across multiple monitoring intervals. We could extend the implementation of IBMon to require monitoring of the VM's original read request (by forcing traps whenever the read call memory page is accessed), or we can monitor and timestamp all work requests, and then match the CQE with the corresponding WQE to get a better estimate of the bandwidth. However, both of these alternatives still include asynchronous monitoring, and hence would not guarantee accuracy, and furthermore are computationally significantly more complex, or, in the case of the first one, even require inserting significant fast path overheads.

Therefore IBMon uses an approach that dynamically tunes the monitoring interval to the buffer size used by the VM. The interval is reduced in the event of small buffer sizes and increased when the VM reads/writes large buffers. Determining that the interval should be increased whenever IBMon determines bandwidth values above the IB limits is intuitive, however it is significantly more challenging to determine the adequate rate at which is should be modified, and when it should be decreased again. Assigning significantly large monitoring intervals, at the data rates supported by IB, will result in significant margins of error of the monitoring process, and will generate excessive amount of CQ entries to be processed in each monitoring iteration. Therefore we rely on the buffer size value, i.e., the *monitoring granularity*, to determine how to adjust the monitoring interval. For very small monitoring intervals the extra memory access can be avoided by using the fabric bandwidth limit as an indication that the interval should be increased. While the current bandwidth estimation mechanism used by IBMon uses this approach, we will further evaluate and compare other alternatives.

3.6 Experimental Results

Next we present the results from the experimental evaluation of IBMon, aimed at justifying the suitability of the design to asynchronously monitor VM's usage of VMM-bypass capable devices such as IB HCAs and the feasibility to attain estimates of such usage with reasonable accuracy.

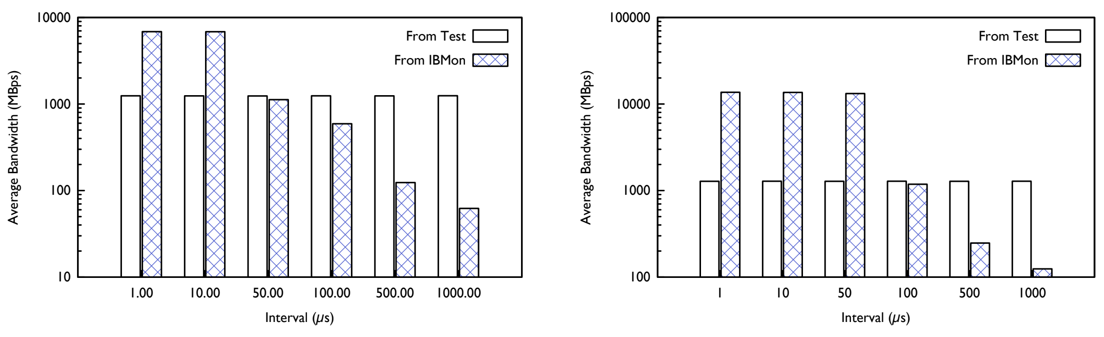
3.6.1 Testbed

The measurements are gathered on a testbed consisting of 2 Dell 1950 PowerEdge servers, each with 2 Quad-core 64-bit Xeon processors at 1.86 GHz. The servers have Mellanox MT25208 HCAs, operating in the 23208 Tavor compatibility mode, connected through a Xsigo VP780 I/O Director switch. Each server is running the RHEL 4 Update 5 OS (paravirtualized 2.6.18 kernel) in dom0 with the Xen 3.1 hypervisor.

The guest kernels are paravirtualized running the RHEL 4 Update 5 OS. Each guest is allocated 256 MB of RAM. For running InfiniBand applications within the guests, OFED (Open Fabrics Enterprise Distribution) 1.1 [100] is modified to be able to use the virtualized IB driver. Both Xen and the OFED drivers have already been modified by us, so as to be able to efficiently support multiple VMs on IB platforms [114]. The workloads are derived from existing RDMA benchmarks part of the OFED distribution.

3.6.2 Monitoring Frequency

The first set of measurements illustrates the relationship between the buffer sizes used by the VM's operation and the monitoring interval for two different buffer sizes. The measurements are gathered by monitoring a single VM and show the bandwidth reported by IBMon and the bandwidth reported by RDMA_Read benchmark executing in the VM. The benchmark performs 20000 iterations of an RDMA read operation for a given buffer size. We vary the monitoring interval from 1 to 1000μs – larger



(a) 64KB (b) 128KB

Figure 4: Tracking Application Bandwidth with IBMon for 64KB and 128KB buffer sizes with different monitoring intervals.

monitoring intervals resulted in excessive amounts of CQ entries.

As seen in the graphs in Figure 4, for very small intervals relative to the buffer size, and therefore the byte count used by IBMon to compute the bandwidth estimate, IBMon reports the maximum bandwidth supported by the IB fabric. This is also used by IBMon to trigger an increase in the monitoring interval. For monitoring intervals which are too large relative to the fabric bandwidth and therefore the buffer size transmitted with each read request, IBMon significantly under performs – at each iteration multiple CQEs are associate with the same, much later timestamp used to then estimate the bandwidth for the total number of bytes, resulting in up to an order of magnitude lower estimates for the largest time interval shown in Figure 4. More importantly, however, the graphs show that in both cases there are interval values for which IBMon performs very well, and only slightly underestimates the actual bandwidth usage by the VM. Using similar data for other buffer size we can either experimentally determine the adequate interval for a range of buffer sizes, which can then be used by IBMon, or compute its bounds based on the fabric bandwidth limits.

3.6.3 Feasibility

Next, for different monitoring granularities, i.e., different buffer sizes and the corresponding interval values, we evaluate the accuracy with which IBMon is able to

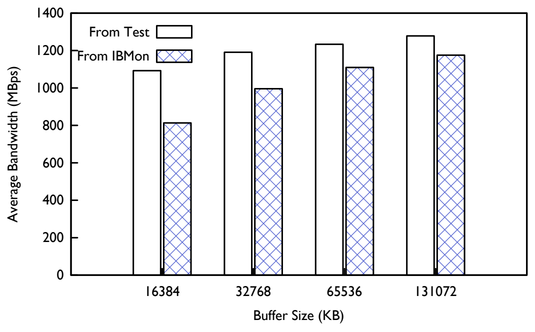


Figure 5: Accuracy of IBMon for different monitoring granularities

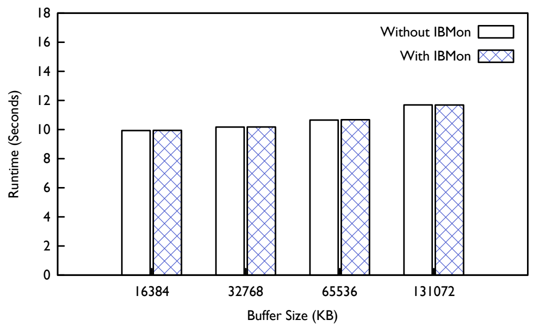


Figure 6: IBMon overheads for different monitoring granularities

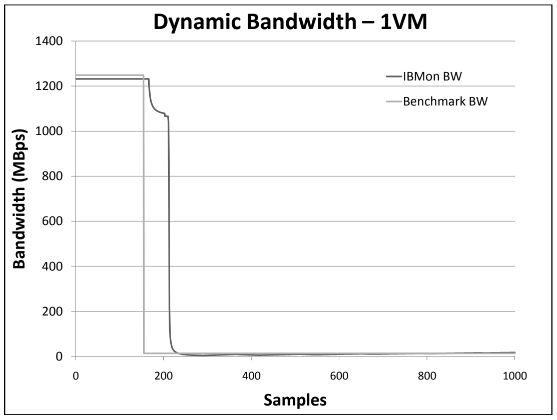


Figure 7: Ability to respond to dynamic changes in the VM behavior

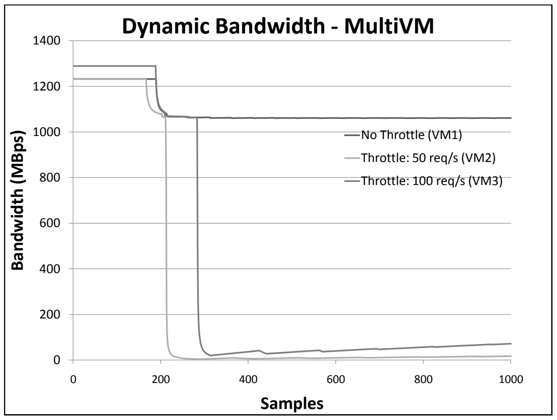


Figure 8: Ability to monitor multiple VMs with different behaviors

estimate the VM's utilization of the device resources. The graph in Figure 5 indicates that at fine granularity, IBMon does face some challenges, and its accuracy is only within 27% of the actual value. However for larger buffer sizes (which is often the case for HPC applications), and therefore longer monitoring time intervals, IBMon is able to estimate the bandwidth quite accurately, within up to 5% for some of the data points in the graph.

3.6.4 Overheads

For each of the monitoring granularities used above, we next measure the overheads of using IBMon. IBMon is deployed as part of dom0, and as such it is executing on a core separate from the guest VM. First, it is likely that in virtualized many-core environments such management functionality will typically be deployed on designated cores. Second, a lot of the processing overhead associated with the current implementation of IBMon is due to the fact that it does have to perform additional computation to estimate the bandwidth, whereas with appropriate hardware support that overhead will be significantly reduced. One overhead which will remain is the fact that IBMon does rely on frequent memory accesses for the monitoring operations, and as such it will impact the application performance due to memory contention. Since the pages are mapped as read-only into IBMon's address space the issue of data consistency will not arise.

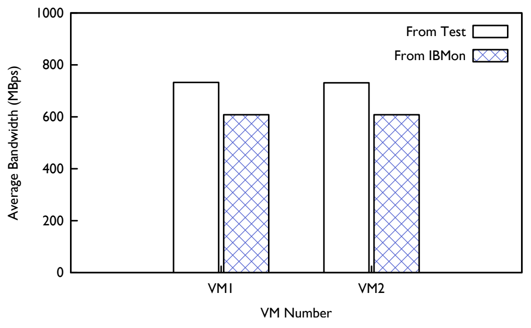
In order to evaluate this impact we compare the execution of the same benchmark with and without IBMon. The results shown in Figure 6 indicate any increase in the VMs execution time is virtually negligible, and that therefore IBMon's overheads are acceptable.

3.6.5 Ability to respond to changes in VM behavior

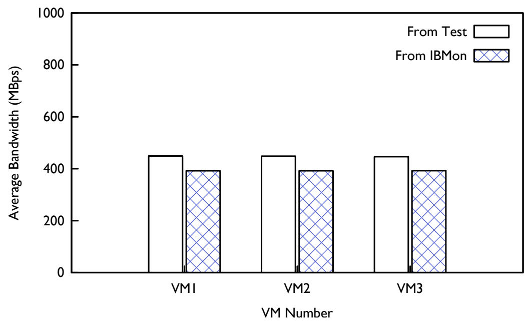
By dynamically monitoring the frequency of the completion events and the buffer sizes used in the RDMA operations, IBMon attempts to determine the monitoring granularity and to adjust the rate at which it samples the VMs' buffer queues. In order to evaluate the feasibility of the approach we modified the RDMA benchmark to throttle the rate at which it issues the read requests and thereby lowered its bandwidth utilization. The graphs in Figure 7 compare the bandwidth measured by the benchmark vs by IBMon and we observe that IBMon is able to detect the changes in the VM behavior reasonably quickly.

3.6.6 Scalability with Multiple VMs

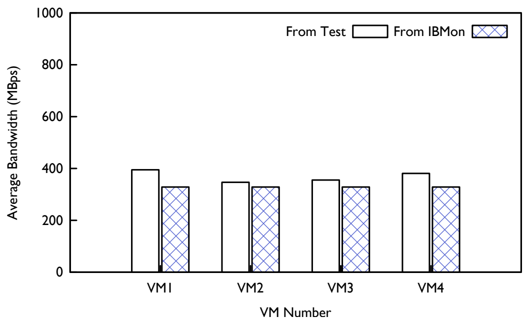
The above measurements focused on evaluating IBMon's ability to monitor a single VM. Next we conduct several experiments involving multiple VMs in order to



(a) 2 VMs



(b) 3 VMs



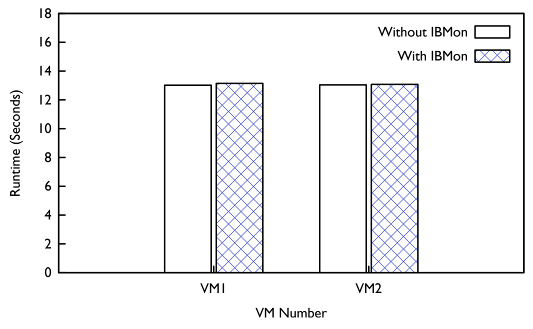
(c) 4 VMs

Figure 9: Accuracy of IBMon in terms of average bandwidth when monitoring multiple VMs

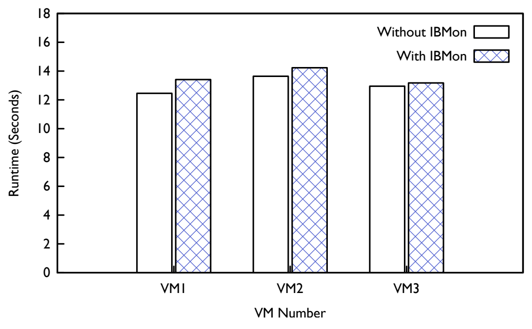
assess the feasibility of concurrently monitoring multiple buffer queues and to track monitoring information corresponding to multiple VMs.

First, the graphs in Figure 9 show IBMon's accuracy in concurrently monitoring multiple VMs. We observe that although IBMon now is responsible for tracking multiple queues and maintaining information for multiple VMs, it is still capable of estimating the behavior of each of the VMs with acceptable accuracy levels. The current estimates are within 15% of the actual value, we do believe that there are significant opportunities in this initial implementation of IBMon to further improve these values.

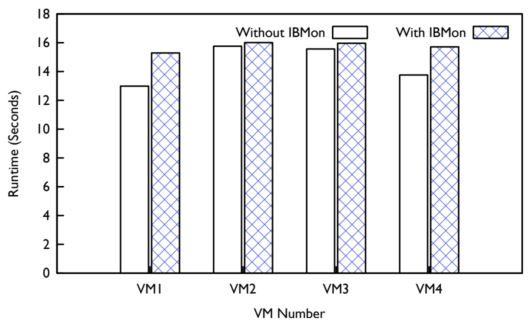
Next, the measurements in Figure 8 show that in the multi-VM case IBMon is able to respond to dynamic changes in individual VMs. For these measurements we use the modified RDMA_Read benchmark and for VMs 2 and 3 we throttle the request rate to a lower value. The measurements do show that IBMon does not respond to



(a) 2 VMs



(b) 3 VMs



(c) 4 VMs

Figure 10: Overheads of IBMon when monitoring multiple VMs

the behavior changes very rapidly, but we believe that evaluating methods to more aggressively tune the frequency interval (e.g., such as those used in AIMD in TCP) will result in additional improvements in its responsiveness.

Finally, the results presented in Figure 10 demonstrate the acceptable overheads of IBMon as it concurrently monitors multiple VMs, again by comparing the benchmarks execution time with and without IBMon. While, as expected, the overheads do increase as we increase the number of VMs, and queues IBMon has to monitor, their impact on the VMs execution is still on average within 5% of the original performance levels.

3.7 Chapter Summary

In this chapter we describe our novel monitoring utility for InfiniBand adapters, termed IBMon, which enables asynchronous monitoring of virtualized InfiniBand devices – an example of VMM-bypass devices heavily used in the HPC community. In the absence of adequate hardware-supported monitoring information, IBMon relies on VM introspection techniques to detect ongoing VM-device interactions and their properties (i.e., data sent/received, bandwidth utilized, or other properties of the data transfers). Such information can then be used for platform management tasks, such as to adjust a VM's CPU or memory resources, or to trigger VM migrations. Experimental results demonstrate that IBMon can asynchronously monitor VMM-bypass operations with acceptable accuracy, and negligible overheads, including for larger number of VMs, and for VMs with dynamic behavior patterns.

While IBMon is implemented for InfiniBand devices virtualized with the Xen hypervisor, its basic concepts and methods generalize to other devices and virtualization infrastructures. Of particular importance are the novel ways in which the virtualization layer asynchronously monitors VM-device interactions that bypass the VMM (i.e., for high performance), and the insights provided regarding requirements for hardware supported monitoring functionality.

CHAPTER IV

FARES: FAIR RESOURCE SCHEDULING FOR VMM-BYPASS INFINIBAND DEVICES

In this chapter, we present our resource management system, FaReS that leverages the I/O usage information from IBMon on communication-intensive applications to provide fairness-based I/O resource allocations. We show a direct correlation between the amount of CPU allocated to the I/O characteristics of a VM. This allows us to dynamically provide fairness guarantees by controlling I/O usage through CPU allocations.

4.1 Challenges for communication-intensive workloads

Parallelism in modern multicore platforms, along with advances in hardware-level support for virtualization, are enabling increased levels of VM consolidation in large-scale data center systems. This implies there will be runtime competition for platform resources, as well as increased levels of dynamism in resource usage. Exacerbated by the spectrum of workloads in the applications being consolidated, these variations can threaten the performance of applications running on shared platforms. An important approach to dealing with this issue is to actively monitor and supervise at runtime [71] the ways in which platform resources are used.

Online monitoring and management take place at multiple levels of data center systems, starting with hypervisor-level fairness and isolation guarantees [16], continuing with rack-level provisioning [50], and extending to data center-level load management for meeting application-level SLAs within global usage or power constraints [95]. We address the hypervisor-level fairness guarantees which provides the basis for effective data center management. In this chapter we use the term fairness to denote the ratio of decrease of application performance when collocated with other applications on the same platform.

Our work addresses high performance backend systems and applications, assuming hardware environments that combine powerful processors with high throughput, low latency system area networks like InfiniBand or 10Gig Ethernet. To efficiently utilize such hardware, modern hypervisors provide methods that permit VMs to directly access these high end communication devices, termed 'hypervisor bypass' in literature [51, 78]. While providing high performance, unfortunately, such solutions also prevent hypervisors from monitoring and supervising device usage by applications. As a result and as shown in this chapter, it becomes difficult or impossible for hypervisors to provide the fairness and isolation guarantees required for successful application consolidation. Specifically, the hypervisor (1) loses its abilities to dynamically gain direct insights into the bypass device usage behaviors of guest VMs, and (2) it cannot prevent device mis- or over-use by some VMs at the potential expense of other virtual machines. The use of static approaches that rely on worst case estimates of VM device usage, of course, would result in low device utilization, which prevents aggressive consolidation and the consequent power and cost savings.

Management of VMM-bypass-enabled Devices. The challenge addressed in this work is how to 'dynamically' manage high performance I/O devices, such as InfiniBand adapters and emerging multi-queue 10GigE NICs. In contrast to traditional devices that are under the complete control of the virtualization software (e.g., via device drivers integrated in the hypervisor [131] or running in a special (driver) domain [16]), for high performance devices like InfiniBand, high performance solutions exploit device-level support to safely map portions of the device's resources into guest VMs, thereby allowing direct VM-device interactions and bypassing the hypervisor, i.e., hypervisor bypass. As a result, the hypervisor is no longer involved on the I/O fastpath, and is used only in setup, termination, and for similar control plane operations. Therefore, it can no longer control the performance impact of the shared use of device resources by multiple VMs.

Table 1: Performance Impact of MG NAS with RDMA Write benchmark.

Config	**RDMA Write**		**MG-4 (MFlops/s)**	**RDMA Write**		**MG-8 (MFlops/s)**
	MBps	Reqs/sec		MBps	Reqs/sec	
Separate	929.86	414.64	2538.25	929.86	414.64	2479.6

Table 2: Performance of NAS benchmarks.

	IS	**CG**
Separated	14.57	4.13
Consolidated	28.48	4.29
Decrease%	95.47	3.87

A hypervisor's inability to supervise application resource usage can adversely affect application behavior, as demonstrated by the measurements shown in Table 1. We perform comparisons between two types of workloads – an RDMA Write benchmark and 2 configurations of the NAS MultiGrid benchmark: MG-4 and MG-8 (where the number shows how many processes are used). In our configuration, the RDMA Write benchmark uses 2VMs (one server, one client), while the MG benchmark is run with a set of 4 VMs. Our platform consists of a pair of Xen hosted servers (more details about the platform appear in Section 4.6). The row labeled 'Separate' shows the performance when each benchmark is run on the platform with no additional load. The row labeled 'Consolidated' shows the performance of the two types of benchmarks when consolidated onto the same platform in a manner that equally distributes all platform resources. The goal, of course, is to fairly share the single platform used by both applications, and to attain this goal, we allocate as much CPU to each benchmark so the degradation in performance is according to the priorities of the VMs. Unfortunately, because the applications share an unsupervised bypass device, they do not actually receive their desired fair shares of the device. Instead, experimental results clearly indicate that the I/O intensive RDMA write VM is able to continue to obtain most of its required I/O resources, thereby 'thrashing' the MultiGrid VM's communications. In fact, the performance of the latter is degraded by almost 40%. In the ideal case, given the same priorities of both applications, they should have seen same performance degradation.

Additional results in Table 2 show that this issue exists not only when one of the VMs executes an aggressive I/O application such as RDMA Write. These results are gathered for 8 VMs running the Integer Sort (IS) and the Conjugate Gradient (CG) benchmarks from the NAS suite, when executing individually vs. consolidated on the same platform, with 4 VMs per benchmark and a VCPU for each VM. When the benchmarks are run on the same platform, the Xen scheduler should load balance the CPUs fairly and the HCA should service each VM in a fair manner. However, we see that IS suffers an almost 50% drop in performance, while CG performance remains almost unchanged. This can be attributed to the fact that the I/O behavior of the CG benchmark interferes with the communications of the IS benchmark. To attain fairness, therefore, the benchmark must be 'throttled' to issue requests in a controlled manner in order to restrict interference between the benchmarks.

FaReS. The *FaReS* approach to Fair Resources Scheduling for VMM-bypass capable devices described and evaluated in this chapter uses:

- *asynchronous, external monitoring* – to continuously monitor all VM-device interactions, externally and without the need to interpose virtualization software into the VM-device fast path, implemented via

- *VM memory introspection* – to interpret the VM's utilization of device resources, and coupled with runtime management that uses
- *cap boost* – to indirectly control a VM's level of device usage and prevent misuse, by dynamically altering the allocations of resources under the hypervisor control, i.e., CPU cap.

High performance is attained because FaReS does not "mediate" all VM-device accesses, but uses a monitoring daemon to asynchronously gather information about device usage. The daemon is not interposed in device communications, either, but uses VM memory introspection to assess current device usage. In this approach, FaReS monitors the state of the VM's memory used by the VM-level device stack and/or the device itself, from which it then infers the device usage data. FaReS then uses this information to adjust the CPU allocations of the VMs sharing the bypass device, using a CPU chargeback mechanism computed from VMs' device usage data. This work applies the approach to InfiniBand devices, but similar methods can be used for any bypass device.

FaReS is implemented for platforms virtualized with the Xen hypervisor, and it is evaluated with RDMA and NAS microbenchmarks, as well as with an MPI-based Map-Reduce application. Experimental results demonstrate that for mixed HPC-related workloads, the use of FaReS degrades application performance modestly, while at the same time, attaining substantial improvements in application fairness. FaReS can provide desired levels of fairness between applications very accurately (within 2%) and achieves this with less than 1% overhead.

Newer bypass devices like InfiniBand may offer hardware improvements that facilitate the implementation of FaReS monitoring and chargeback. First, concerning monitoring, there will likely be device-level counters useful to the hypervisor for computing realistic device usage data. Such counters could be used to better determine application-level message or request rates (note that each application-level message may be comprised of multiple device-level requests). Second, next generation InfiniBand devices already enforce prioritization and isolation via hardware-supported Virtual Lanes. While these allow control over physical link allocations, the number of Virtual Lanes as well as the number of physical Queues for Multi-device Queue Devices are limited, thereby also limiting VM consolidation in terms of the number of VMs' communications mapped to these Virtual Lanes/Queues. The FaReS approach can be used to remove such limits.

The remainder of the chapter is organized as follows. In Section 4.2 we discuss certain additional techniques that are used by FaReS. In Section 4.3, using experimental data, we discuss the impact of various types of VM I/O behaviors on the performance of other concurrently running workloads, and we establish guidelines for detecting and controlling such I/O thrashing. Section 4.4 provides an overview of how FaReS interacts with hypervisor and IBMon. Section 4.5 describes the chargeback mechanisms used in FaReS. Experimental evaluations are discussed in Section 4.6. We present a summary of the chapter at the end.

4.2 Xen Credit Scheduler

At its core, FaReS utilizes the memory introspection capabilities of IBMon and the memory management APIs described in Chapter 3. The additional techniques used by FaReS to control VM allocations are mentioned here.

The Xen scheduler [140] provides an interface to set the CPU values for the required VMs. We use this interface with the IBMon tool to set the value. Internally, each VM is assigned credits at its startup (100). These credits are deducted every clock tick (10ms) and the VM is preempted at the scheduling interval (30ms). Depending on whether the credits are positive or negative, the VM's priority is set to *Under* or *Over*. The scheduler will first try to run the VMs in the respective VCPU runqueue in the *Under* priority. If no such VM is present, it will run a VM in the *Under* priority from another VCPU runqueue followed by the local VCPU runqueue in *Over* priority and finally by the remote VCPU runqueue in *Over* priority. When the CPU cap value is set for a VM, the credits are assigned accordingly. We interact with the *XenStat* library to get VM related information as well as to set the CPU caps.

4.3 Understanding I/O Thrashing in RDMA-enabled Devices

To control the utilization of CPU and I/O resources, we must be able to determine at runtime the extents to which such resources are used. For CPUs, counters offered by hardware and accessible to hypervisors continually assess resource usage. For high end I/O devices like InfiniBand, however, devices asynchronously carrying out VMs' requests do not provide hypervisors with detailed insights into their operation. In fact, our previous work with IB devices has shown that in the absence of accurate device-level timestamps, it is impossible to accurately determine the I/O bandwidth utilized by a VM at any given point [110]. Similarly, while it may be possible to determine average I/O (bandwidth) utilization at some time scale, such assessments, when made on the wrong time scale can be misleading and will lead to inadequate resource management actions. We next use several examples to better characterize the relationship between various properties of the I/O behavior of a VM, their impact on the performance of other co-located workloads, and the utility of using such properties to represent trends in the VM's bandwidth utilization. The goal is to then use these insights, by monitoring changes in those multiple properties of the I/O accesses of a given VM, to infer information regarding the VM's bandwidth and device usage.

Buffer Size. Figure 11 shows the performance of the RDMA Write benchmark for two different application behaviors. These two application instances are run between two VMs each (a client and server for each application). Here, both VMs have the same amounts of resources (CPU, memory, I/O). We compare two cases: one in which

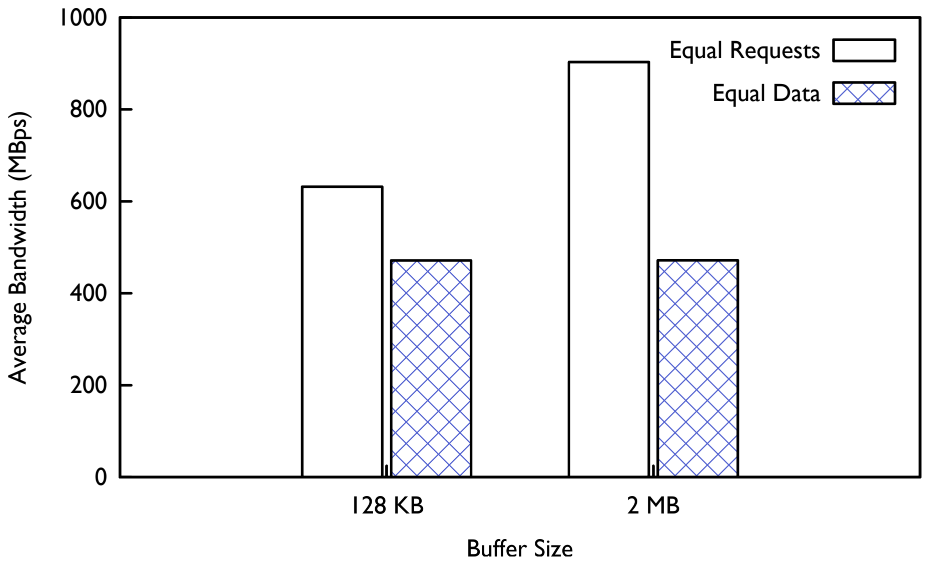


Figure 11: Average RDMA Write BW for different application behaviors.

Table 3: Request issue, completion times for different buffers with CPU caps.

CPU Ratios	**100:100**		**75:75**		**50:50**	
Buffer Size	2KB	2MB	2KB	2MB	2KB	2MB
Issue Time (s)	0.034	21.44	0.044	21.5	0.103	21.61
Average BW (MBps)	448.44	932.67	445.74	926.34	188.77	927.01
Total Time (s)	1.52	22.93	2.05	23.51	3.11	24.54

both VMs send the same number of requests (10K) and a second one in which they send the same amount of data (number of requests times the buffer size). In the first case, the VM with a 2MB buffer size prevents the VM with the 128K buffer from reaching maximum bandwidth, thereby 'thrashing' it. In the second case, the 128KB VM issues 160K requests to send the same amount of data as the 2MB VM. This results in the reverse problem, where the 2MB VM gets thrashed by the 128KB VM.

When applications with smaller buffer sizes are used jointly with applications with larger buffer sizes, measurements presented in Table 3 make apparent that the 'small buffer' VMs may get thrashed. Here, both VMs issue 10K requests, where issue time is the amount of time spent in posting requests and checking for completions. The figure shows that for the small buffer (2KB) VM, the issue time is much less than for the 'large buffer' (2MB) VM. This is because for the 2KB VM, the actual completions of requests may occur before the checking of completions has been performed. This is due to the asynchronous nature of InfiniBand and the lack of hardware timers for timing completed requests. When the CPU cap is decreased, the effect becomes more pronounced for the 2KB VM, because it may complete more or less the same number of requests in the scheduling interval but may not also get time to check for completions in that interval; this has a negative effect on its issue time. For the 2MB VM, it can still issue almost the same number of requests, which better exploits the DMA ability of the HCA and hides the fact that the VM is not always scheduled to run.

From these measurements, it is clear that a device operating asynchronously 'on behalf' of a VM cannot be assumed to naturally emulate with its behavior the VM's processing behavior enforced by the hypervisor's CPU scheduler. To attain fairness and isolation for VMs' device usage, therefore, we must find ways to control the data transfers performed by different VMs and/or ways to prevent such transfers from affecting each other, as per VMs' priorities. To do so without requiring new device capabilities or changing device drivers, we choose the 'indirect' method of control by which we penalize VMs that make excessive use of the device in terms of their ability to run and issue additional device requests. Naively, for an application to issue X number of requests in an interval, it needs Y% of the CPU. By decreasing the CPU to Y/2, we expect that the number of requests issued would decrease by some appropriate fraction (X/2 in the linear case).

Transmission Depth. Figure 12 demonstrates the effect on bandwidth when the CPU allocation is changed along with changes in the transmission depth for the benchmark. Here, by transmission depth we mean the number of requests issued in a

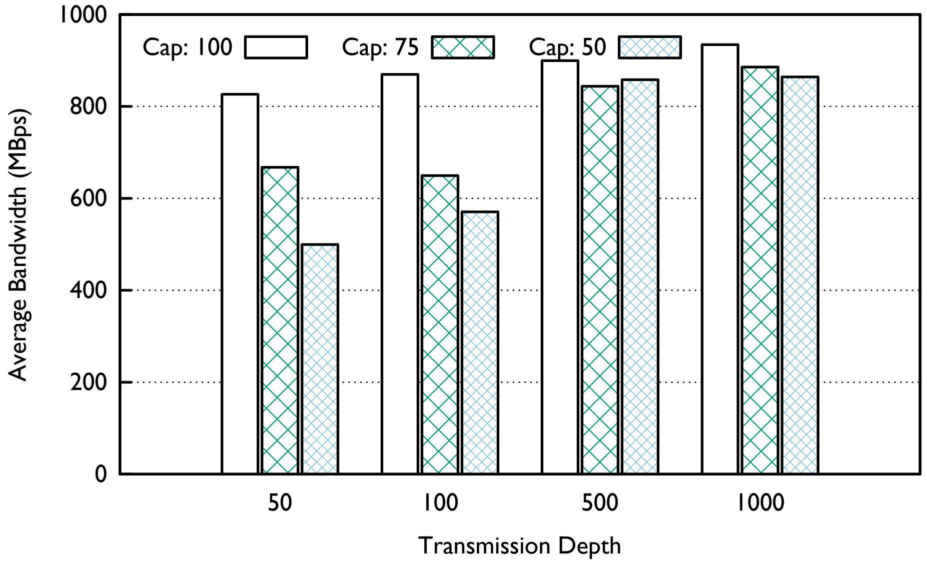


Figure 12: Effect on Average BW by Transmission Depth and CPU Caps

single loop without checking for completions. The figure shows that even when little CPU is allocated, for a large enough transmission depth, the application's average bandwidth is nearly equal to the Peak bandwidth. We attribute this to the size of the Xen scheduling time slice (*10ms*), which is large enough for the application to post these requests. Since the data transfer happens 'outside' the VM world, it is not included in the scheduling accounting, thus giving the application much higher bandwidth. For smaller transmission depths, the check for completions may happen in the next schedule phase, and the application infers that the small datasize takes the whole timeslice to transfer, resulting in a lower bandwidth. Lowering the CPU cap effectively translates to decreasing the time slice, resulting in a much lower bandwidth. The peak bandwidth reached across all configurations is about 932 MBps.

For each request issued in a loop, there is a corresponding completion entry written in the memory by the HCA. As the transmission depth is increased, the number of entries increase and may span multiple pages. By monitoring these pages we can infer device utilization changes. Then we use this information to adjust CPU caps.

Communication vs. Computation Ratio. To demonstrate the potential impact

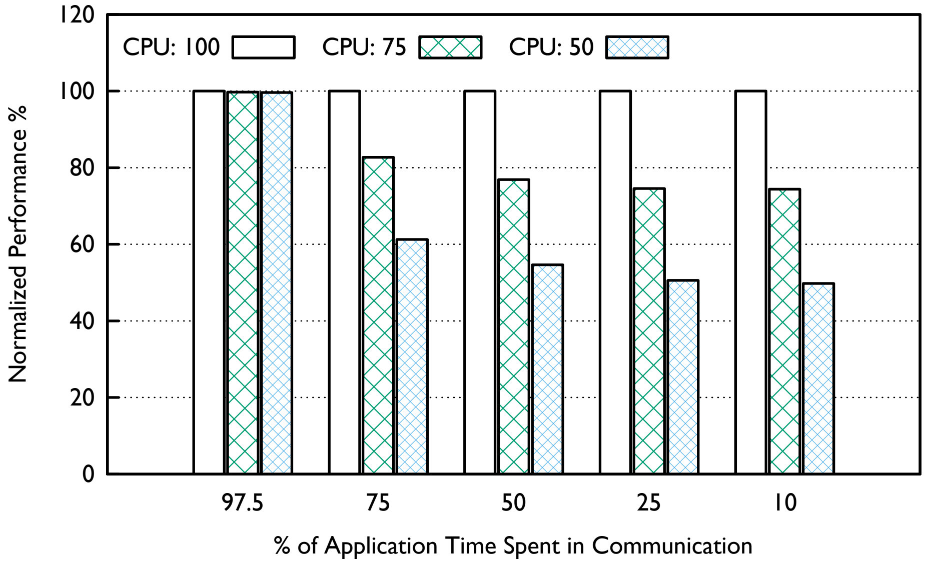


Figure 13: Variation in Application Performance with different Communication and Computation ratios under CPU cap changes.

of the VMM scheduler, we modify the existing RDMA Write benchmark to include a MD5 computation. After the RDMA Write is performed, the client computes a MD5 hash on the data in the buffer before sending it. We choose the MD5 hash function as it is simple to include in the test as well as it scales with the size of the input data thus varying the compute times for the benchmark.

In Figure 13, we observe that depending on the amount of computation vs. communication, the application's sensitivity to changes in the CPU cap differs. The percentage in the graph corresponds to the time the application spends in posting sends as well as polling for completions. The remainder of the time is the computation time for MD5. We subtract the setup time for the QPs, CQs, etc. from the total time. For decreasing percentages of communication we increase the number of MD5 hash computations in a linear fashion. For the percentage of 97.5 we do not use the MD5 hash. The figure shows that as the communication percentage decreases i.e. computation to communication ratio increases the effects of the VMM scheduler on the performance of the application/VM is more visible. The performance decrease matches the decreasing CPU cap for the VM.

By understanding the application's communication or computation ratio, we can infer the impact of the reduction of the CPU cap on the performance of the application/VM. Depending on the priority of the VM, we dynamically adjust the caps for the VMs. For example, a low priority communication intensive VM will get much reduced CPU cap, since we know that the impact on the VM performance will not be large. On the other hand, for a compute intensive high priority VM, maximum CPU will be allocated depending on the current system load.

Summary of Observations. The above experiments show that an application's bandwidth may not be dependent solely on its CPU usage. Other application-level parameters, like number of completed requests and buffer size, also affect the application's performance. When determining the amount of CPU to be allocated to an VM, we therefore also use the buffer size and 'fullness' of the entry page as additional indicators concerning the utilization of the HCA by the VM. Depending on the VM's communication/computation ratio the above information is used to affect the VM's CPU usage by different levels of the entry page. Specifically, we charge time back' to the VM, where the amount charged back depends on its buffer size (as motivated by the measurements shown above). We term the resulting algorithm the 'Cap Boost' method, discussed in detail in Section 4.5.

4.4 FaReS Overview

Figure 14 represents the key components of the FaReS software architecture, more specifically, our current prototype implementation of FaReS for platforms with InfiniBand I/O devices, virtualized with the Xen hypervisor. In order to monitor the VM-device interactions, FaReS relies on a monitoring utility - IBMon [110]. IBMon relies on dynamic introspection of the guest VMs' memory used by their OFED stacks

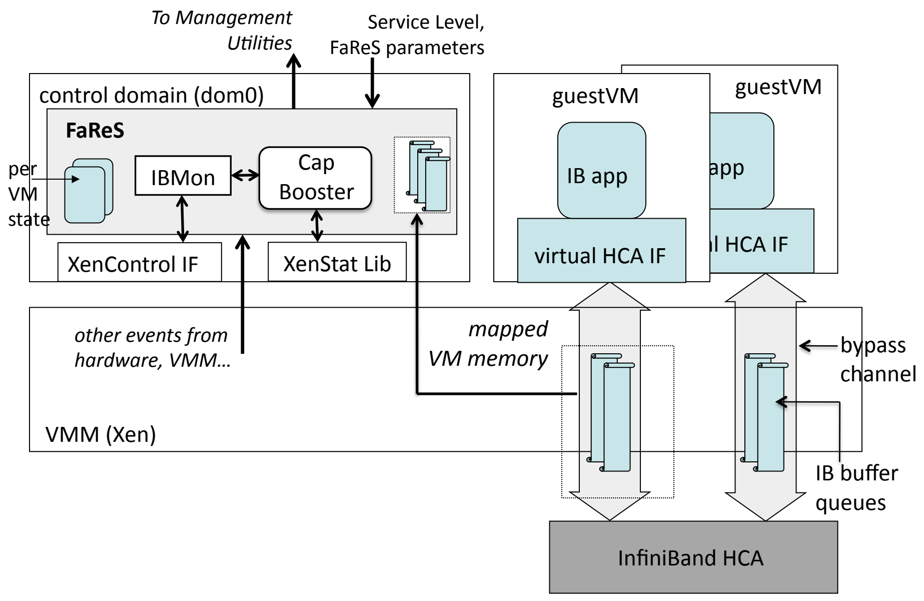


Figure 14: FaReS Software Architecture

to asynchronously extract information regarding their I/O usage, and is further described in Chapter 3.

The ‘Cap Booster’ component of FaReS is responsible for controlling the CPU allocations for a particular VM, based on information provided by IBMon, along with additional inputs regarding existing system loads, domain information (e.g., priorities, past resource usage, etc.). This component uses the concept of ‘Cap Charge-Back’, where the VM’s CPU cap is reduced when it sends too much data. This algorithm is presented in more detail in Section 4.5.

FaReS relies on the hypervisor-level APIs for interacting with the resource management entities on the virtualized platforms. For our prototype realization of FaReS, these include the XenControl and XenStat libraries. The XenStat library is used to get/set the CPU allocations and domain information. The XenControl library is used to map the pages from the guest VM to FaReS for monitoring. FaReS can also be augmented to take more external inputs from system administrators to tweak the performance of the system depending on the policy required. FaReS has certain parameters that can be changed, to make scheduling more aggressive or more conservative. Also, it takes the various priorities of the VMs as input to provide different levels of service. FaReS relies on a set of parameters that can be modified to make the scheduling more aggressive or more conservative. In the latter case, the primary objective of FaReS is to manage the performance degradation experienced by VMs as a result of consolidation with other VMs, at potentially different priority levels, as opposed to aggressively reduce the resources allocated to the existing workloads, so as to potentially permit the collocation of other VMs or to attain other benefits, such as reduced energy requirements.

FaReS interacts with IBMon (Chapter 3) to determine the current number of requests completed by a running VM. FaReS uses this in its Cap Booster component to determine how to adjust the CPU allocation for the VM. FaReS maintains a 'per VM state' to keep track of requests completed, current CPU usage, CPU cap assigned. Also, FaReS provides an interface to higher level management tools for retrieving the monitored data and can be used to couple the monitoring information with large-scale 'cloud' management utilities.

4.5 FaReS Cap-Boost Scheduling

Cap-Boost Scheduling tries to provide certain VM-level guarantees to maintain a fair share of resources. We attempt to proportionally share the platform resources between contending applications/VMs with the objective of maximizing the performance of the higher proportioned application while maintaining a lower-bound performance for other resource-contending applications. The proportion can be decided by system administrators and forms an input to the algorithm. We define 'Fair Share' for such

Algorithm 1 Algorithm for FaReS Cap-Boost Scheduler.

```
1: Start with unbounded resource allocation for all VMs
2: IBMon provides the Requests Completed (reqs) and Time required to send 1
   request (t1req)
3: Use different levels of fullness of CQ page (High Level - HL, Medium Level - ML,
   Low Level  LL)
4: function FindCPUBoost(reqs, t1req)
5:     if $reqs > HL$ then
6:         $cap\_boost = \alpha * reqs * t1req$
7:     else if $reqs > ML$ and $reqs < HL$ then
8:         $cap\_boost = \beta * reqs * t1req$
9:     else if $reqs > LL$ and $reqs < ML$ then
10:        $cap\_boost = \gamma * reqs * t1req$
11:    else if $reqs < LL$ then
12:        $cap\_boost = \delta * (MAX\_REQs - reqs) * t1req$
13:    end if
14:    return $cap\_boost$
15: end function
16:
17: Current_load = Get_Running_CPU_percent from xenstat
18:
19: if $VM = highpriority$ and $Current\_load < MAX\_LOAD$ and $reqs > HL$ then
20:     $cap\_boost = MAX\_REQs * t1req$
21: else if $VM = lowpriority$ then
22:     $cap\_boost \leftarrow$ FindCPUBoost(reqs, t1reqs)
23: end if
24:
25: Add cap_boost to current VM cap
```

VMM-bypass enabled systems by regulating the amount of degradation perceived by the VM. In our algorithm the VM degradation is inversely proportional to the share of resources given to the VM – i.e., a higher share of resources (credits) will result in lower degradation. This share of resources is the resource proportion (RP), which is allocated by the administrators. For example, if we allocate a RP of X:Y for the VMs V1 & V2 respectively, we expect that for every X% performance reduction in V2 there is Y% performance decrease in V1.

This section describes the 'Cap Boost' Algorithm 1 that we have used in FaReS. The algorithm is a scheduling engine which takes as inputs the VMs' communication (completed I/O requests), computation (CPU credits and caps), proportionality requirements, system parameters (CPU loads and bandwidth), and outputs resulting (adjustments to) VMs' CPU allocations.

In order to account for the I/O requests the algorithm uses a 'charge-back' mechanism, where it penalizes the VM for sending an excess of data beyond its allocated proportion. We claim that a charge-back mechanism is appropriate for such VMM-bypass devices since the CPU/OS/VMM is not responsible for the data transfer, but instead, it is performed by the HCA (in case of IB). Thus, it also could be said that the VM effectively 'gains' time by not participating in the data transfer. By 'charging back' this CPU amount to the VM we aim to throttle the request rate of the VM.

The amount of 'charge back' also depends on the resource proportion assigned to the VMs, where a VM with lower resources will be penalized more for sending excess data. The algorithm determines this excess data by scanning the CQ for number of requests completed. It uses weighted parameters called 'Scalers' for different values of excess data. These 'Scalers' denoted by α, β, γ, δ exist for different numbers of completed requests in the CQE page. They can be positive or negative which results in the appropriate boost. The algorithm uses these parameters to tune the decrease in CPU 'charge back'. For example, a VM with very large number of CQEs needs to be charged back more in order to reduce its request rate. We determine the value for these 'scalers' empirically for a given set of VMs. The algorithm also uses another parameter called 't1req' to calculate 'charge back'. The 't1req' parameter depends on the buffer size and corresponds to the time required to send a request. Also, the 'Scalers' allows us to tune the aggressiveness of the algorithm as more CPU can be reclaimed from the VM.

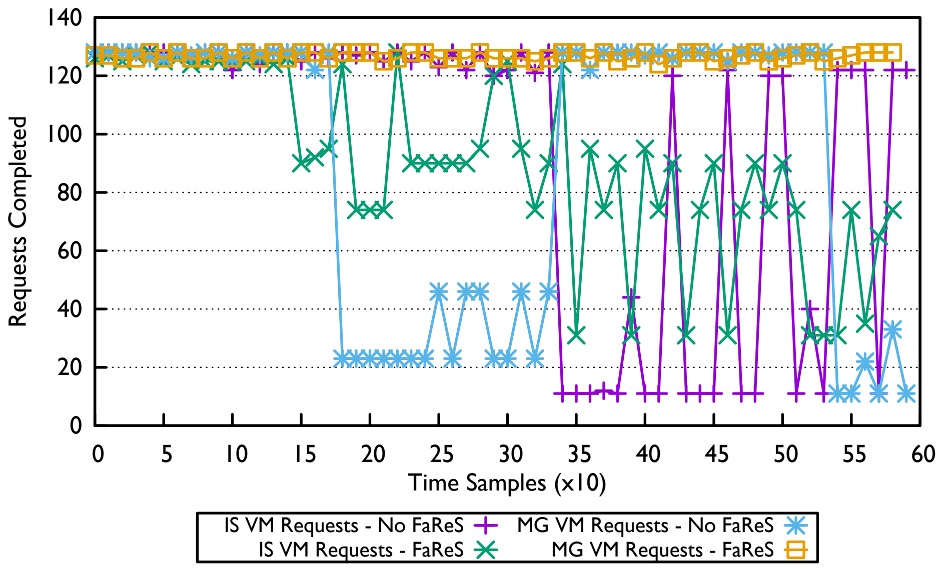


Figure 15: Requests Completed change for High/Low Priority VMs with FaReS.

4.6 Experimental Evaluation

The testbed consists of two Dell PowerEdge 1950 servers. One server has dual-socket quad-core Xeon 1.86GHz processors, while the other has dual-socket dual-core Xeon 2.66Ghz processors (in total 12 cores). Both servers have 4GB of RAM. Mellanox MT25208 HCA (memfull cards) cards are used in both the machines which connected via a Xsigo VP780 I/O Director Switch. We are using a recent Xen 3.3 unstable distribution on both the servers. Also, both the servers are running the same para-virtualized Linux kernel running in dom0. The InfiniBand modules we used in [114] have been updated to run under the new Linux 2.6.18.8 kernel and Xen 3.3. The guest OS' are configured with 512 MB of RAM and have the OFED-1.1 distribution installed. This distribution has been modified to run inside the guest VM. We are also running the OFED-1.1 distribution in the dom0s. We use OpenMPI version 1.1.1 in all our VMs as well as dom0s.

To demonstrate the utility and effectiveness of FaReS we evaluate the system on a mixture of RDMA-based benchmarks, NAS Parallel Benchmarks [92] and a sample application based on the MPI Map-Reduce paradigm.

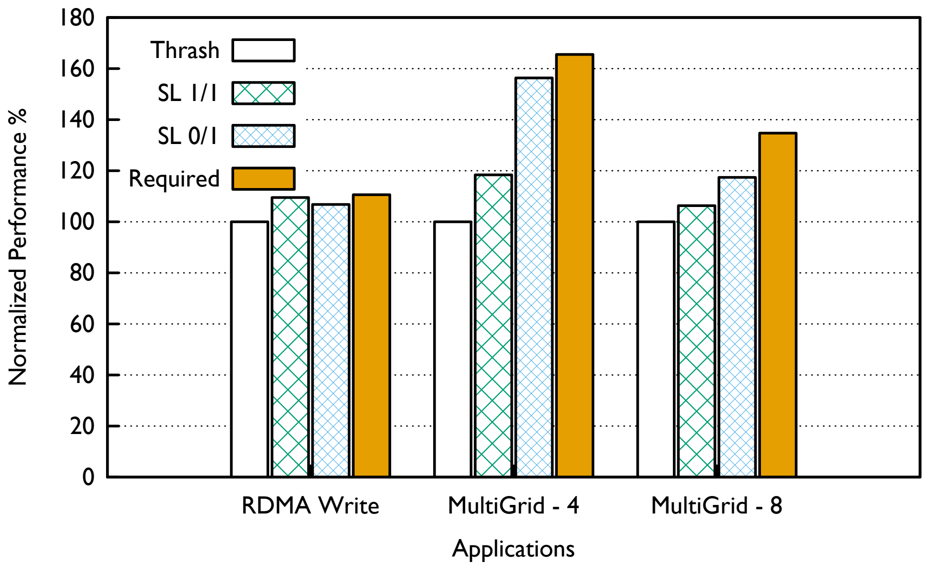


Figure 16: Normalized Performance of RDMA Write and MultiGrid with FaReS.

4.6.1 FaReS Behavior

Figure 15 shows the change in the requests completed for low and high priority VMs with and without FaReS running over continuous time samples. The samples are collected for a low priority VM part of Integer Sort (IS) benchmark set of VMs and a high priority VM part of the MultiGrid (MG) benchmark VMs. All the other VMs which are part of the running benchmark exhibit similar behaviors. The figure shows that when FaReS is not running the MG benchmark does get thrashed by the IS benchmark and has a low request completion rate. With FaReS there is considerable improvement in the requests completed for the MG benchmark, which remain constant throughout the experiment while the requests of IS are completed at a slower rate.

4.6.2 Application Performance

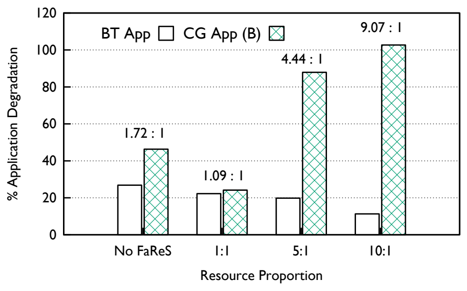
The next set of measurements demonstrate FaReS' ability to control the impact of consolidation and limit the performance degradation experienced by VMs to levels which are proportional to their relative priorities.

RDMAWrite vs. MG. In Figure 16, we show the MultiGrid and RDMA Write benchmarks running in multiple VMs. The RDMA-Write benchmark is configured to run in 2 VMs (1 server, 1 client). The MultiGrid-4 and MultiGrid-8 are 2 different configurations with 4 and 8 processes we ran for the benchmark in 4 VMs. We compare the performance of the application with and without FaReS in place with the original performance ("Required") when the benchmark is run by itself with no additional load. FaReS is not involved in the baseline "Thrash" configuration, and all VMs compete for resources. We use two cases to illustrate how FaReS affects the application performance with different priorities.

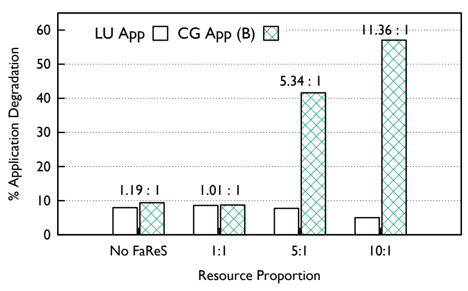
In the SL (Service Level) 1/1 case FaReS manages the two benchmarks, both executing at high priority and FaReS tries to provide an equal set of resources. Even for this case, we observe improved performance compared to the baseline case. Since FaReS can '’throttle' the I/O requests and thereby affect when the request reaches the HCA. The HCA is not inundated with requests as is with the non-FaReS case and has lesser number of outstanding requests. This results in FaReS providing a degree of overlap of communication and computation phases of the VM.

For SL 0/1 the RDMA Write benchmark is run at low priority and FaReS will aggressively try to reduce resources for it. We observe that the performance of the high priority MG benchmark is improved significantly more compared to the baseline case, and that the performance of the low priority RDMA write benchmark is affected more compared to the SL 1/1 case.

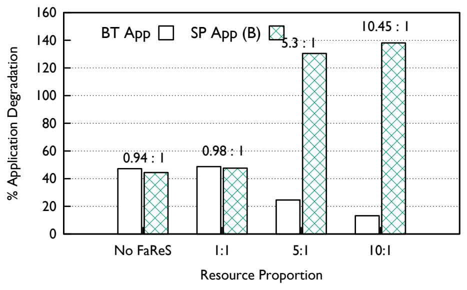
NAS Benchmarks Performance. Figure 17 shows the performance of various pairs of NAS benchmarks when run in a consolidated manner without FaReS as well as with FaReS for different ratios of Resource Proportions (1:1, 5:1, 10:1). The application marked as (B) is run as a background application and is assigned the lower set of resources. Here the background application is run continuously till the main application finishes. The labels on the bars indicate the performance degradation ratio



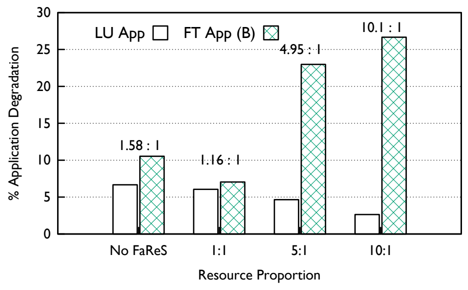
(a) BT vs. CG



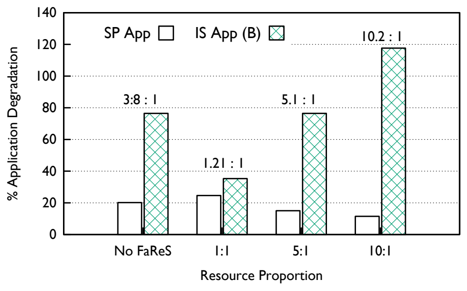
(b) LU vs. CG



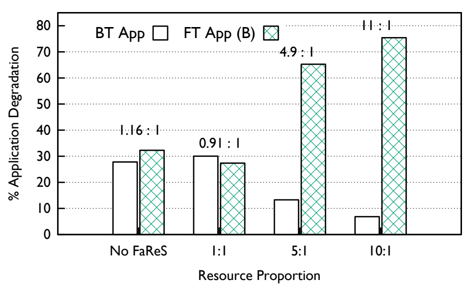
(c) BT vs. SP



(d) LU vs. FT



(e) SP vs. IS

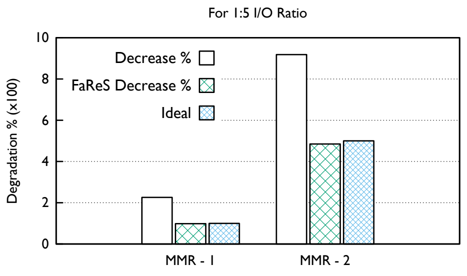


(f) BT vs. FT

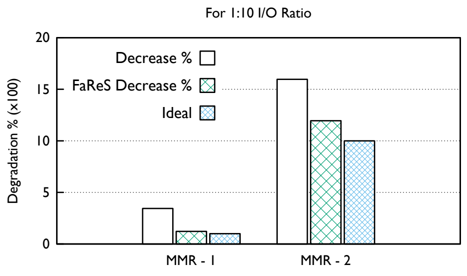
Figure 17: Performance of NAS Pairs of Application with FaReS and Different Resource Proportions. The labels on the bars represent the Resource Proportion achieved.

between the background application and the main application. The performance here is the execution time for the application. We can see from the figure that using a 1:1 resource proportion degrades each application by the same amount. In the case for the different resource ratios FaReS shows on average does 5:1, 10.36:1 with a standard deviation as 0.33, 0.8 for the 5:1, 10:1 ratios respectively.

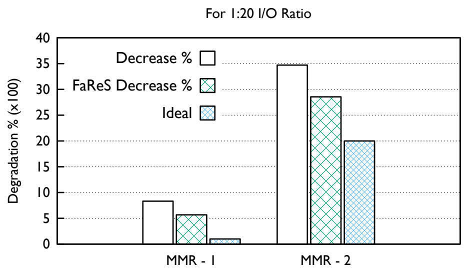
FaReS Overhead. For each of the benchmarks used in the above experiments, we also gathered measurements with FaReS running to just monitor the set of VMs without perform any CPU scheduling. We found that the change in performance with



(a) FaReS Ratio = 4.92:1



(b) FaReS Ratio = 9.8:1



(c) FaReS Ratio = 5.03:1

Figure 18: Performance of Pair of MPI Map-Reduce Applications with FaReS

respect to the consolidated values is less than 1% across all the sets of applications. This shows that there is minimal perturbation to the running application due to FaReS.

4.6.3 Map-Reduce Application

In order to evaluate the different levels of fairness provided by FaReS when instances of the same application exhibit different behaviors we use an application which is a subset of a larger HPC application, the Gyrokinetic Toroidal Code (GTC)[68]. The GTC is a 3-Dimensional Particle-In-Cell code used to study micro-turbulence in magnetic confinement fusion from first principles plasma theory. The subset application is based on MPI Map-Reduce and performs sorting of the GTC particle data. We call this application MPI Map-Reduce (MMR). The idea of fairness here is to provide relative decrease in performance between applications according to the I/O or Compute Ratio and to minimize the negative effects of consolidation as mentioned in Section 4.3.

Figure 18 shows the performance for 2 different instances of MMR running on the platform with and without FaReS. We evaluate the performance of the instances for 3 different I/O ratios 1:5, 1:10, 1:20. The base case used here is for both applications sending same number of requests (1:1). The ratios are the amount of I/O messages the instances issue. Each instance is run on a set of 4 VMs (2 on each physical machine). We scale the I/O for MMR-2 in this case. We run the MMR-1 instance multiple times while the MMR-2 instance is running and provide an average performance decrease. The ideal case is when both applications incur the same performance decrease as the I/O ratios. For the 1:20 ratio we increase for MMR-2, the communication-to-compute ratio to 80%. In this case we see that FaReS has little effect on the fairness ratio since MMR-2 outputs a large amount of I/O and behaves similar in the case of the RDMA Write and MG Grid test.

We see from the figure that both instances suffer more than twice the ideal performance(in case of MMR-2 the decrease is 1600% for 1:10 I/O ratio). When FaReS is enabled we see that performance improves for both instances when compared to the non-FaReS case. This is due to the ability of FaReS to overlap the communication and computation phases for the application instances as explained in the above section.

4.7 Chapter Summary

In this chapter we describe FaReS, an approach for fair resource scheduling in virtualized platforms with high-end, VMM-bypass capable devices. In order to address the lack of hypervisor involvement in the VMs' I/O accesses in these environments, FaReS relies on (1) asynchronous monitoring – to continuously monitor all VM-device interactions, (2) VM memory introspection – to interpret the VM's utilization of device resources, and (3) a 'cap boosting' technique – to dynamically control the allocations of those resources which are under the hypervisor control, i.e., CPU caps, so as to prevent or boost the VM's level of I/O utilization.

Based on observations about the characteristics of the I/O behavior of a VM which impact the performance of other collocated VMs, and the sensitivity to CPU allocations exhibited by such I/O behaviors, we asynchronously extract information regarding the VMs' I/O behavior and use it to make appropriate adjustments in the VMs' CPU allocation, using a 'charge-back' mechanism which penalizes a VM for sending an excess amount of data. We claim that a charge-back mechanism is appropriate for such VMM-bypass devices since the CPU/OS/VMM is not responsible for the data transfer but it is the HCA in case of IB. Thus, it also could be said that the VM effectively 'gains' time by not participating in the data transfer. FaReS is implemented for InfiniBand platforms virtualized with the Xen hypervisor and evaluated using microbenchmarks and realistic applications. We show that FaReS assigns fairness to mixed workload HPC applications within 2% of the required value for 3 different levels of fairness. For all such applications FaReS provides a very low (<1%) monitoring overhead.

CHAPTER V

RESOURCE EXCHANGE: LATENCY-AWARE SCHEDULING IN VIRTUALIZED ENVIRONMENTS WITH HIGH PERFORMANCE FABRICS

While FaReS is designed to provide fairness guarantees to communication-intensive workloads it cannot provide any guarantees to latency-sensitive applications. In this chapter, we show that in order to achieve guarantees for latency-sensitive we have go beyond the bandwidth-based metrics that FaReS uses and enable more finer-grained resource provisioning.

We do this by introducing a Virtual Currency Abstraction using Resos, which enables us to dynamically map the virtual I/O resources to underlying physical resources through Economic methods like Congestion Pricing. We use these methods to build multiple policies that allow latency-sensitive applications to meet the specified guarantees.

5.1 Latency-sensitive applications in virtual environments

Virtualized infrastructures have seen strong acceptance in data center systems and applications, but have not yet seen adoptance for HPC codes which require I/O to arrive within predictably and consistently short durations (i.e., no 'noise'), in exchanges like ICE, CME, and NYSE which need trades to complete in a certain amount of time (deadlines), also in server systems performing phone call switching or multimedia delivery, which require soft deadlines to be met. At the same time, however, there is a need for virtualization in such environments, as demonstrated by recent work in which soft deadline capabilities are added to the Xen hypervisor in order to support VOIP call switching server applications [74]. In fact, there are even greater opportunities for virtualization for exchanges, since their average machine utilization can be less than 10% under normal load conditions, due to conservative provisioning in order to meet peak demands and high availability requirements. There are also opportunities with HPC codes, for which data centers routinely report issues with under-utilization despite intensive user of batch job schedulers, and these opportunities will grow as leadership machines move to the exascale, due to the difficulties many programs will have to efficiently and concurrently use resources at that scale.

In this chapter, we develop mechanisms to better enable hosting and collocation of latency-sensitive applications, such as those listed above, on virtualized platforms. We are specifically targeting high-performance RDMA-enabled interconnects, for two reasons. First, these devices have features such as higher bandwidths ($\geq$10Gbps), as with InfiniBand [52] and 10GigE [91] network fabrics, support for remote DMA, protocol offload, and kernel bypass, and, as a result, they provide the low-latency high-bandwidth requirements needed by the aforementioned types of applications. Second, the same set of features also enables efficient near-native I/O virtualization of these devices [108], which makes them more suitable candidates for virtualized platforms for latency-sensitive applications.

However, to enable consolidation for latency-critical applications, problems to be solved go beyond efficient I/O virtualization, to also include dealing with the shared use of I/O, memory, and compute resources, in ways that minimize or eliminate interference. Newer generation InfiniBand cards allow controls such as setting a limit on bandwidth for different traffic flows and giving priority to certain traffic flows over others, thereby controlling latency for that flow. Also, what is required is for the hypervisor scheduler to be enhanced to make sure that it can take into consideration the latency requirements of the VMs, the loads they pose on the shared network infrastructure, and the levels of interference and noise they cause due to their I/O access patterns. This is particularly challenging in virtualization solutions for RDMA devices such as InfiniBand HCAs, since due to the VMM/OS-bypass capabilities provided by these devices, the hypervisor loses control in monitoring and managing device-VM interactions. Therefore, it becomes harder for the hypervisor to maintain data rate or latency service levels to its guest VMs.

To address these issues, we develop a hypervisor-level solution which dynamically adjusts the compute and I/O resources made available to each VM, so as to provide better isolation – i.e., control noise – and improve performance for latency-critical applications, while still allowing consolidation and improved aggregate resource usage, without requiring worst-case-based reservations. Our approach, termed *Resource Exchange* or *ResEx*, leverages (i) IBMon – a tool developed in our prior work [110] that enables dynamic monitoring of the I/O usage for virtualized InfiniBand devices, and (ii) Virtual Currency Abstraction that uses congestion pricing models and supply-demand microeconomic concepts to provide more flexible and fine-grained resource allocations.

Specific contributions of our research include (1) the design and implementation of the *Resource Exchange* approach, and (2) its resource trading 'currency' abstraction – *Resos*, and (3) mechanisms for dynamic monitoring of VMs' resource usage and the interference, or the 'congestion' they cause, so as to be able to (4) support range of resource pricing policies derived from our Virtual Currency Abstraction. (5) Using Nectere – a latency-sensitive benchmark modeled after a financial trading exchange, (6) we implement and experimentally evaluate two such policies, and demonstrate the feasibility of the ResEx approach to make RDMA-based virtualized platforms more manageable and better suited for hosting even latency-sensitive workloads.

The remainder of this chapter is organized as follows. Section 5.2 uses experimental data to show the effect of collocation on latency-sensitive applications. In Section 5.3 we give a brief description of the economics pricing theory we rely on.

Section 5.4 describes the latency-sensitive benchmark that we have developed and use throughout the chapter. In Sections 5.5 and 5.6 we describe in detail the design goals and implementation for ResEx. Section 5.7 provides experimental results demonstrating the benefits of ResEx.

5.2 Collocation Impact on Latency-sensitive workloads

While high-end fabrics, such as the InfiniBand fabric considered in our research, exhibit high bandwidth and low latency, their shared usage for latency-sensitive workloads has been limited. When running a latency-sensitive workloads on shared network infrastructure it is expected to observe some level of perturbation in the achieve latency. Under some conditions, the variability may be within acceptable levels, and/or may be exhibited only by some of the collocated applications, and not by the most latency-sensitive ones. In other cases, however, this variation can exceed desired limits, especially when the degree of interference with other applications is high.

Our goal in this section is to demonstrate that, in spite of the superior performance properties of high-end fabrics like InfiniBand, their shared usage by latency-sensitive applications may result in prohibitive levels of variability. For this, we explain the changes experienced by a low latency workload when it is being run with (Interfered) and without (Normal) another interfering workload. We use different configurations (i.e., different data sizes) of the Nectere latency-sensitive benchmark developed by our group (explained in Section 5.4) as both a low latency workload and second as an interference generator, each run in a separate VM. Both benchmark instances, i.e., both VMs, are assigned to separate CPUs, therefore there is no response-time variability due to CPU scheduling.

Figure 19 shows the frequency distribution of the low latency workload when it is run with and without the interference load. In the Normal case the latencies are highly stable at around 209 μs. But when it is run alongside the interfering load the

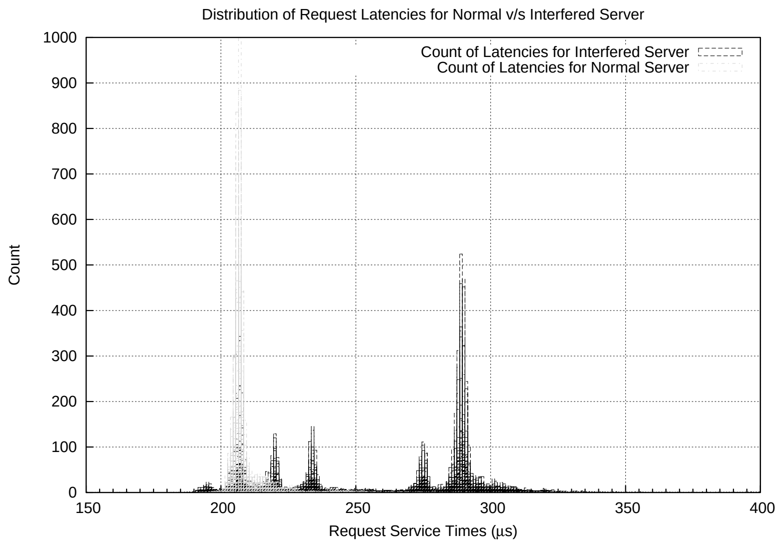


Figure 19: Distribution of latencies of a normal server versus an interfered server with additional load.

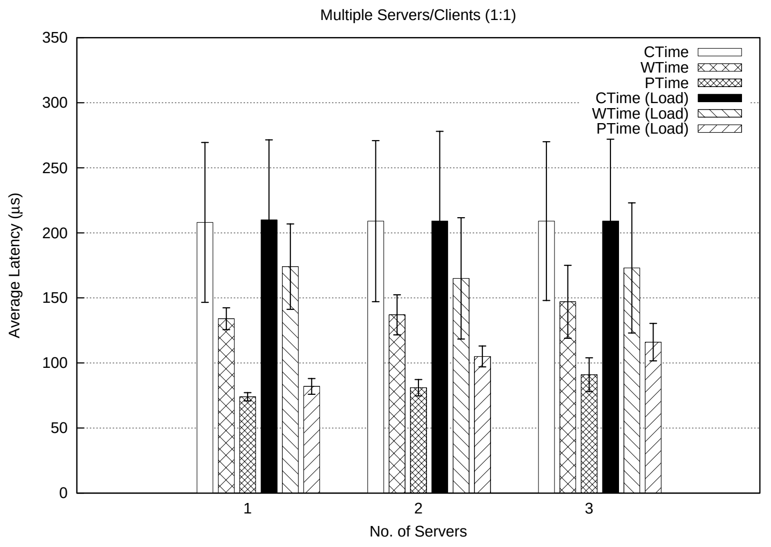


Figure 20: Change in server latency for multiple servers with interfering load.

latencies are distributed across the interval. This shows that not only the average increases but the variation/jitter as well even for RDMA-based interconnects. Note that for certain requests the service time is smaller than Normal server possibly due to no interference occurring for those requests.

Figure 20 shows the change in server latency as server and client count is increased, as well as when there is interfering load added to the system. There are three parts to the server latency - Compute Time (CTime), I/O Wait Time (WTime), Polling Time (PTime). Each of them is shown with the variation in error bars. Here, the number of servers refers to separate instances of the low-latency workload configuration of the benchmark. For each group of servers, we report average latency when these are the only collocated applications on the node, vs. when also run with a VM running an interference generator. Again, all VMs are allocated to their own CPUs. Since CTime is independent of I/O interference it remains fairly constant. From the figure we observe that both WTime and PTime start increasing with load since now a RDMA operation from the server takes more time to process due to the interfering load at the device level. We also observe, however, that when collocating only the VMs running the original application, the interference effects, and any latency degradation, are much less noticeable, demonstrating that it is feasible to allow latency-sensitive workloads to share the virtualized platform.

5.3 Economics and Congestion Pricing

ResEx relies on certain properties of the underlying fabric – InfiniBand – and hypervisor – Xen, in our case. While the InfiniBand properties and hypervisor extensions are described before in Chapters 2-4, here we describe the economics and congestion pricing ideas used in ResEx.

Many of ideas that we incorporate into ResEx have been motivated by principles of microeconomics like supply and demand [47]. By supply we mean the set of physical resources that are available to each VM to use in a given time period or ‘epoch’. The demand is the amount of resources that each VM consumes in that ‘epoch’.

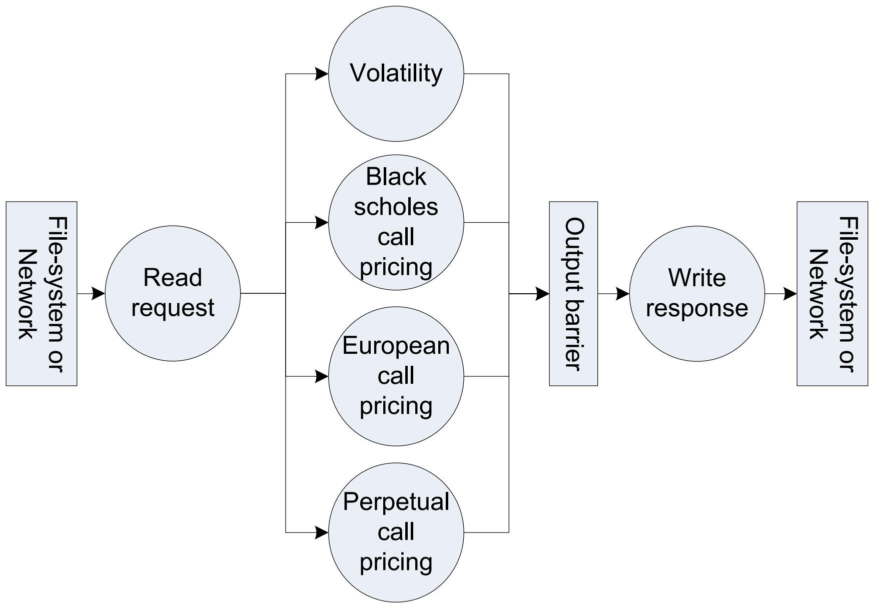


Figure 21: A Financial Application Configuration for Nectere.

Microeconomics theory states that when the demand for a commodity goes up the price for that commodity increases and the converse is true as well. We use this idea as the main motivator for reducing congestion on shared resources by VMs.

This idea is more expressly presented in another concept of economics derived from supply and demand called *'Congestion Pricing'* [138]. By increasing the price of a resource we aim to reduce the demand for it thereby reducing the 'congestion' for that resource. Congestion arises since the demand or 'resource usage' exceeds the supply or 'resource availability'. Changing the price for this resource corresponds to charging the heavy users for causing the congestion. We describe this in more detail in Section 5.5.3.

5.4 Nectere - A Latency-Sensitive Benchmark

In order to evaluate the behavior of latency-sensitive workloads on virtualized RDMA-enabled fabrics, we develop an RDMA-based latency evaluation benchmark, which allows us to easily parameterize and experiment with this class of applications. The benchmark, termed Nectere [45], is modeled after a commercial trading engine used by one of our industry collaborators (ICE – Intercontinental Exchange) [126]. The benchmark uses a server-client model similar to trading systems where multiple clients post transactions and request feeds from a trading server hosted by the Exchange, and it includes traces which model the I/O and processing workloads present in an exchange like ICE today. Furthermore, the benchmark further allows us to easily configure it and change its I/O behavior – e.g., rates, sizes, etc. – or per-request processing times, thereby simulating applications with different performance and resource usage profiles. It also allows us to combine various financial computations to mimic workloads. Figure 21 shows one possible financial configuration for Nectere.

In Nectere, our clients send multiple transaction requests to the server using RDMA, each of which is time stamped by the client. The server maintains these requests in a queue and processes them in a first come first serve basis. The FCFS criterion is important to exchanges since each transaction may change the outcome of the next one and we implement the same criteria for our benchmark.

In addition to trace data, since we do not have the proprietary data and processing codes used in real exchanges, our server uses a financial processing algorithm library [19] for request processing to perform operations such as Black Scholes Options Pricing. The clients wait for the server results, timestamp the reply, and calculate the latency of the request by taking a difference between the two timestamps.

Nectere also provides an online monitoring interface to an external agent, running inside each VM, through which it can continuously report the observed server-side latencies. The agent may then forward this information to the main ResEx module running in Dom0, as described in Section 5.6.

5.5 Design Principles of ResEx

In order to provide a better understanding of our implementation of ResEx, we next describe the methodologies leveraged in its design. First, we describe how we can track InfiniBand I/O usage of VMs and therefore the I/O interference between VMs. Second, we introduce our 'Virtual Currency Abstraction' called Resos that provides a dynamic mapping of the impact of CPU allocation to IB resource usage. The dynamism comes from the allocation of Resos as well as price updates using the 'Congestion Pricing' model which we describe next. Finally, we outline the various goals of our resource pricing schemes and discuss alternatives on charging VMs.

5.5.1 Tracking I/O Usage and Interference for VMM-Bypass InfiniBand Devices

Since the VMs can directly talk to the devices it is almost impossible for the hypervisor to estimate I/O usage for such devices. We use our IBMon tool [114] to track the amount of IB resources that a VM is consuming. By using IBMon we can estimate the amount of data that a VM sent in a given interval of time. This amount of data corresponds to a number of packets that the HCA has to send on behalf of the VM. Each packet size is equal and is referred to as a MTU (Maximum Transmission Unit). By knowing the size of the MTU we can infer the number of packets that a VM sent. Therefore, we track the 'number of MTUs sent' (MTUSent) by a VM as a metric for I/O usage.

Each application within a VM uses a buffer size which may be different for each application. We call the ratio of an application's buffer size to another application's as the 'buffer ratio' (BR). We also use this metric for estimating the amount of I/O performed by one VM compared to another.

We define 'interference' as the positive change in any I/O latency or a negative change in device usage perceived by the VM or the application within the VM. First, a direct way to track interference is to get feedback from the application in terms of the latency it perceives from its requests. Second, an indirect way, would be to infer from the IB memory how many requests were completed by the HCA on the VM's behalf in a given time interval and then decide the percentage of bandwidth used by a

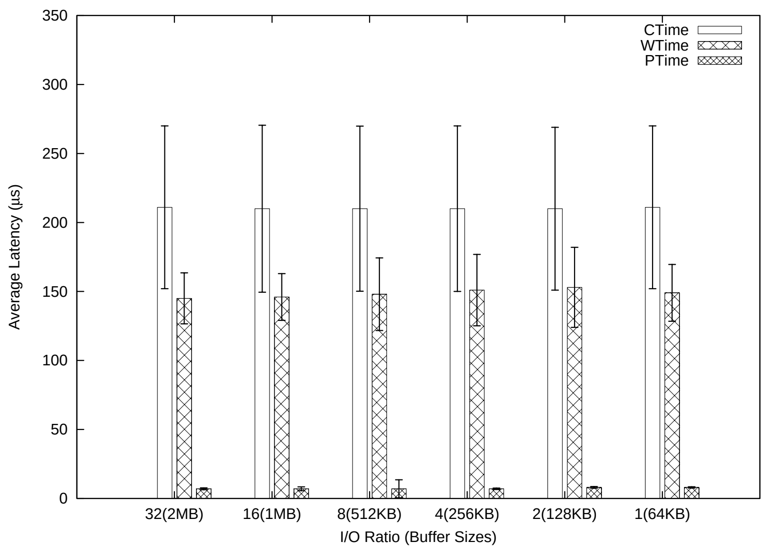


Figure 22: Histogram of server latency with different I/O ratios between interfering and normal servers.

VM. By knowing the percentage of bandwidth consumed we can estimate its impact on other VMs. Once we can estimate the impact of one VM on another's latency we aim to control it using CPU scheduling or capping the interfering VM's CPU. We discuss those ideas next.

5.5.2 Relationship between IB I/O Latency, Buffer Ratio and CPU Cap

In order to determine how to best control I/O latency interference across VMs, we perform several experiments to establish the dependencies between various parameters of the VM's I/O and CPU usage.

Figure 22 shows the change in I/O latency for a VM when another interfering VM is running. We run our Nectere application within each VM. Each instance has a different application buffer size. We report latencies for the server side of the benchmark. The number in brackets corresponds to buffer size of the interfering VM. The number before the brackets is the 'buffer ratio' of the 'interfering VM' with respect to the 'reporting VM'. The buffer size of the reporting VM is set to 64KB. We

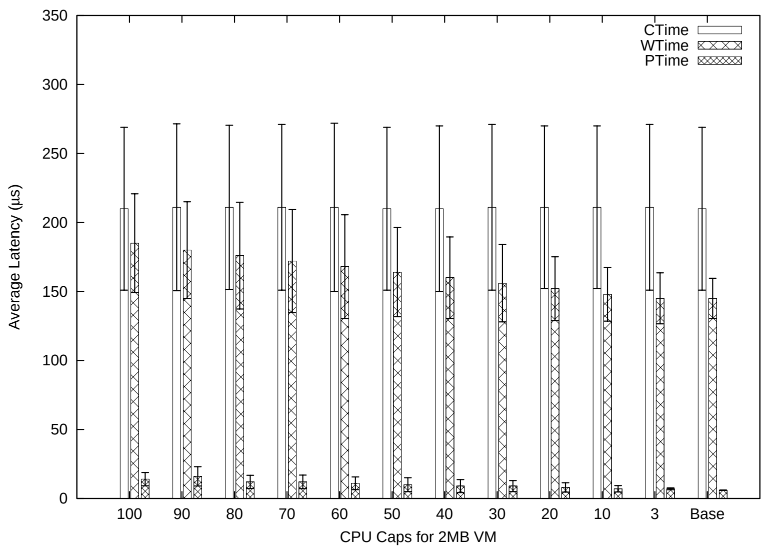


Figure 23: Change in normal server latency as CPU cap is decreased for interfering VM.

compare latencies achieved by the reporting VM when the CPU cap of the interfering VM is set according to the buffer ratio. For example, the CPU cap for a 256KB VM is set to 100/4 = 25%. Since the latencies experienced by the reporting VM do not change between all the instances we can say that the CPU Cap has a direct relationship with the buffer ratio and I/O latencies experienced by a VM.

In Figure 23 we set the buffer size of the interfering VM to 2MB and compare the latencies of the reporting VM when the CPU cap for the interfering VM is set accordingly. We see that by changing the CPU cap steadily the latencies experienced by the reporting VM decrease and when the CPU cap is equivalent to the buffer ratio-based value the latency experienced is equal to the base latency.

Therefore, by knowing the buffer sizes of the VMs and I/O latency experienced by the VM we can identify an interfering VM and assign a particular CPU cap in order to remove congestion. We use this as an indicator for our ResEx algorithm as described in Section 5.6 in order to provide a consistent CPU allocation for the VMs.

We base the consistent CPU allocation by unifying the resources used by a VM under a single entity which we describe next.

5.5.3 Virtual Currency Abstraction

This model is responsible for identifying dynamically the impact of resource allocations like CPU Capping on the resources used by the VMs as well as the impact on latency-sensitive applications. We provide the abstraction by introducing the concept of 'Resource Units' or *Resos* using which VMs 'buy' resources to use during their execution. Here, as 'buy' we refer to the ResEx ability to deduct Resos from the VM, based on its resource usage. Each Reso enables the VM to buy a certain amount of CPU and IB MTUs; sending more data will result in deduction of more Resos. The total number of Resos in a system is determined by the entire set of available physical resources in the system – the 'supply'. This set, in our case, comprises of the InfiniBand bandwidth link and CPU Cycles. We first determine the aggregate available resources, i.e., the corresponding aggregate Resos, and then distribute this equally among all collocated VMs. The Resos can also be distributed unequally, e.g., based on priority of the VMs. VMs can be charged for their resource usage differently, depending on the level of interference they cause, and based on the goals of the pricing algorithm (see the following section). Summarizing, Resos is a mechanism to abstract different types of resources and their availability for each VM.

Resos are deducted at every interval for a VM based on its I/O and CPU usage, as described in Section 5.5.1. A collection of intervals is called an 'epoch' and after every epoch we replenish the number of Resos of a VM to the original allocated value. Any Resos left over from the earlier epoch are discarded. Deduction of Resos depends on the objectives of a pricing system. These objectives and pricing schemes are discussed in the section below.

5.5.4 Goals of Pricing

In general any pricing scheme should be able to regulate the supply and demand of the commodity. Therefore, when the price for a commodity is increased the demand for that commodity should fall. We use the same notion of pricing in ResEx when charging VMs for resource usage. In our current work we consider 2 goals for our pricing strategies, outlined below.

5.5.4.1 Maximize Resource Utilization

The pricing scheme should be able to allow VMs to purchase resources whenever they can and have enough Resos' to do so. Here the scheme will have fixed prices for a unit resource and is identical for all VMs. This allows each VM to use its resources to the maximum, and at a rate depending on the application(s) running within the VM. Once the VM does run out of Resos, its resource usage should be kept either to a minimum or completely blocked until the next epoch when its Resos may be regenerated. In this scheme, resource prices are set at the start of each epoch uniformly for all VMs, based only on the aggregate availability of and demand for resources.

5.5.4.2 Lower Latency Variation

Another goal of a pricing scheme is to allow each VM to pay for resources differently. The pricing differentiation starts when a VM starts using more of one resource causing other VMs to suffer in their performance – i.e., causes interference. The interference causes an increase in latency of the applications, which in turn may miss their SLA for delivery/receipt of data. Thus, interfering VMs will need to be charged more in order to reduce their demand for the resource and reduce congestion.

These and other pricing strategies require support for, dynamic monitoring of resource availability and usage (CPU and I/O resources in our case), as well as for determining the 'interference' or 'congestion' caused by one VM, so that it can be charged for its resources in a differentiated manner, based on some priority notion or other policies.

5.6 *ResourceExchange Implementation*

In this section we describe in detail the implementation of ResourceExchange or ResEx approach to lower I/O congestion for VMs using InfiniBand devices. First, we define how the Resos are distributed between all the VMs and the initial charging policy. Next, we describe 2 pricing policies that we developed based on the set of goals outlined at end of the previous section.

5.6.1 Charging VM Resource Usage in Resos

Since we are dealing with a multi-resource setup, Resos simplifies the way to charge VMs on resource usage. We first break down the CPU and I/O resources into indivisible quantities which can then be charged 'by the Reso'. ResEx performs allocation of Resos at every epoch, which in our case is 1 second. Resos for the resources consumed are deducted at every interval of 1 millisecond.

5.6.1.1 CPU Charging

ResEx converts the CPU cycles based on the frequency to a set of Resos for each CPU depending on the epoch/interval time. It then associates a VM with a CPU. We currently allocate an entire PCPU to each VM, thus the VM can use the entire CPU and the set of Resos associated with it. It charges every percentage of CPU the VM consumes in the interval. This percentage corresponds only to the CPU cycle count for the interval. Initially, the rate of charge is 1 Reso per CPU percent. Therefore, the VM is initially allocated $PercentPerInterval * NumberOfIntervals = 100 * 1000 = 100,000$ Resos for CPU consumption. Here the CPU percent is the indivisible quantity used for charging.

5.6.1.2 I/O Charging

ResEx allocates Resos based on the capacity of the InfiniBand. The physical capacity of our IB Link is 8Gbps (due to the 8b/10b IB encoding) i.e. 1 Gigabyte per second. From Section 5.5.1 we use MTUs as the smallest chargeable quanta. We assume a default MTU size of 1024bytes(1KB) used by applications to send data. Therefore, the IB Link is capable of sending $LinkBW/MTUSize = 1*1024*1024*1024/1024 =$ 1048576 MTUs in one second or in one epoch. Initially, we charge 1 Reso per MTU sent. Here the entire Resos are shared between the VMs since the link is shared between them. ResEx can adjust this sharing to be equal between the VMs or unequal based on priority or weight of the VM.

We have now broken down each physical resource into their chargeable units. Each of these units can be charged using a single currency called Reso. Next, we describe how ResEx implements different charging policies based on the goals described in Section 5.5.4.

5.6.2 FreeMarket & Maximize Resource Utilization

In this scheme, every VM is charged based on the amount of I/O and CPU resources consumed. The rate of charging remains same for all VMs. For every interval and monitored VM, ResEx detects the number of MTUs sent by the VM (using IBMon) and the CPU cycles consumed (using the XenStat library). This is converted to a number of Resos to be deducted from the VM's allocation. Now, this value is deducted from the VM's Reso allocation. Thus, using this scheme every VM can use their maximum set of resources allocated in the epoch and this scheme achieves the maximum resource utilization pricing goal. We call this pricing scheme FreeMarket to denote that the VMs can freely purchase their resources.

However, when the VM's remaining Resos is below a certain limit (10% in our case), and more than 10% of the epoch is remaining, we start decreasing the CPU for that VM gradually. The CPU is decremented by 10% from its earlier allocated value. We do this to ensure that we do not have to abruptly stop the CPU for the VM when it runs out of Resos. This simply ensures a gradual decrease in performance experienced by the VM rather than a sudden stoppage. There are multiple ways in order to reduce the CPU when the VM runs out of Resos but those are beyond the scope of this thesis.

Algorithm 2 shows the pseudo-code for the pricing scheme. The charging calls use the same rate of 1 Reso per unit resource for converting the resources used to Resos. The CPU cap returned depends on the number of Resos left of the VM allocation and is changed as described above.

Algorithm 2 FreeMarket Algorithm

```
 1: InitializeResos(FreeMarket)
 2: while true do
 3:     for all MonitoredVMs do
 4:         IBMTUs ← GetMTUs(ThisVMId)                          ▷ Use IBMon
 5:         CPUPct ← GetCPUPercent(ThisVMId)
 6:         IBResos ← ConvertToIBResos(IBMTUs)
 7:         CPUResos ← ConvertToCPUResos(CPUPct)
 8:         CPUCap ← GetCPUCap(ThisVMId, IBResos, CPUResos)
 9:         DecrementResos(ThisVMid, IBResos, CPUResos)
10:         SetVMCap(ThisVMId, CPUCap)
11:     end for
12: end while
```

5.6.3 I/O Shares & Lower Latency Variation

For meeting the second goal of pricing resources we now incorporate any latency feedback from the application within the VM in charging Resos. We base this charge on the Virtual Currency Abstraction where the VM(s) causing the congestion are charged more. Algorithm 3 describes the pseudo-code of the pricing scheme, where we now adjust the charging rate for the interfering VM when we detect that a VM is experiencing increased latencies. The line [6] in Algorithm 3 computes the I/O interference percentage for the current VM. It looks at the latencies reported by the VM and computes the average and standard deviation of the values. The percentage increase in either of these is then returned to the scheme. If the percentage increase is greater than a certain value (i.e., SLA guarantee) then the interfering VM is found and its charging rate is increased based on the following formula:

$$Increase\ In\ Rate(r') = IOShare * IntfPercent$$

where IOShare is define as:

$$IOShare = \frac{MTUsSentByInterferingVM}{TotalMTUsSentByVMs}$$

Next, we compute the CPU cap to be computed for the current VM, decrement the Resos used and set the VM cap. When the interfering VM is being monitored then it will be charged with the new rate as computed in the reported VM in the last iteration of the loop. The CPU cap now for this interfering VM will be set as

$$New\ CPU\ Cap = \frac{100 * PreviousRate}{PreviousRate + r'}$$

5.7 *Experimental Evaluation*

The testbed consists of two Dell PowerEdge 1950 servers. One of the servers has dual-socket quad-core Xeon 1.86 Ghz processors, while the other one has dual-socket dual-core Xeon 2.66 Ghz processors. Both servers have 4GB of RAM. Mellanox MT25208 HCA (memfull cards) cards are used in all machines, connected via a Xsigo VP780 10Gbps I/O Director Switch. We are using a Xen 3.3 unstable distribution on all servers. Also, all servers are running the same para-virtualized Linux kernel running in dom0. We are using para-virtualized InfiniBand driver modules which work under the Linux 2.6.18.8 dom0 and domU kernels. The guest OS' are configured with 512 MB of RAM and have the OFED-1.2 distribution installed. This distribution has been modified to run inside the guest VM. We are also running the OFED-1.2

Algorithm 3 I/O Shares Algorithm

```
1:  InitializeResos(IOShares)
2:  while true do
3:      for all MonitoredVMs do
4:          $IBMTUs \leftarrow$ GetMTUs(ThisVMId)                         ▷ Use IBMon
5:          $CPUPct \leftarrow$ GetCPUPercent(ThisVMId)
6:          $IOIntfPct \leftarrow$ GetIOIntf(ThisVMId)
7:          $IntfVMId \leftarrow$ GetIOIntfVMId(ThisVMId)
8:          $IOShare \leftarrow$ GetIOShare(ThisVMId, IntfVMId)
9:          $ChargeRate \leftarrow$ ChangeIBRate(IntfVMId, IOIntfPct)
10:         $IBResos \leftarrow$ ConvertToIBResos(IntfVMId, IBMTUs, ChargeRate)
11:         $CPUResos \leftarrow$ ConvertToCPUResos(IntfVMId, CPUPct, ChargeR-
    ate)
12:         $CPUCap \leftarrow$ GetCPUCap(ThisVMId, IBResos, CPUResos)
13:         DecrementResos(ThisVMid, IBResos, CPUResos)
14:         SetVMCap(ThisVMId, CPUCap)
15:     end for
16: end while
```

distribution in the dom0s. Each guest domain is assigned a VCPU each in order to minimize the effects of shared CPUs.

We use certain terminologies in the experiments below which we explain first. We refer to an application running within a VM by its configured buffer size. For example, 64KB VM means that the Nectere application uses 64KB as its buffer size. Interfering VM is a VM that has a buffer size greater than 64KB. The base case refers to a configuration where only one instance of the benchmark is run without an interfering load. In most of the experiments we use 2 VMs each on 2 separate physical machines unless otherwise mentioned. One of the VMs is the reporting VM and the other is the interfering VM. The latencies are always mentioned for the reporting VM.

To demonstrate the utility and effectiveness of ResEx we use different combinations and configurations of the Nectere Latency benchmark. The results are classified by the goals and types of the pricing system developed.

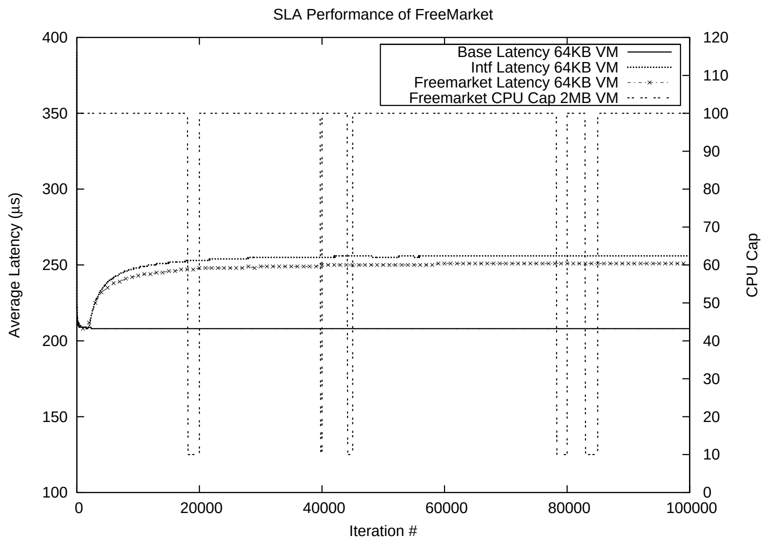


Figure 24: Comparison of application latency when using FreeMarket algorithm.

5.7.1 Maximization of Resource Utilization

Here we compare the application performance of the FreeMarket algorithm with the base case and the case with interfering load. Figure 24 shows the change in the application latency when the FreeMarket algorithm is used. We can see that the latency of the 64KB VM (reporting VM) is lower when FreeMarket allocation is performed than the interfering case. This is due to the fact that the CPU cap is lowered for the 2MB VM periodically whenever its Reso count decreases below a minimum. This achieves some more control over the requests issued by the 2MB VM and thus reduces latencies for the 64KB VM.

Figure 25 shows the deduction in Reso when the FreeMarket algorithm is used. The algorithm keeps deducting Resos until a minimum level (10%) is reached after which it starts reducing the CPU Cap. The effect of this is seen by the 2MB VM. It also 'maximizes resource usage' by always allowing a VM access to the resource if it has enough Resos.

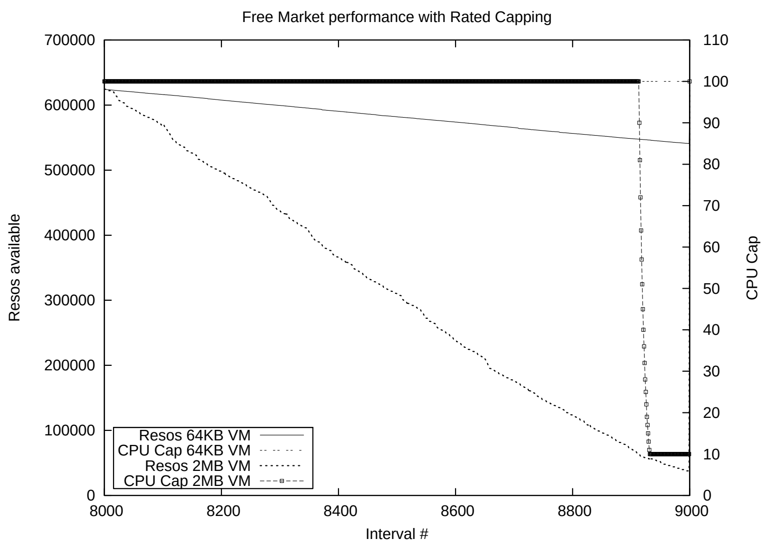


Figure 25: Change in Reso during FreeMarket algorithm.

5.7.2 Lower Latency Variation

In Figure 26 we compare the application performance when the IOShares algorithm is used. We see that the algorithm is able to achieve near base case latencies for the application by taking into consideration the interference percentage of the 64KB VM and thus 'charging' the 2MB VM more for resources used. The CPU Cap is changed dynamically to a lower value for the 2MB VM by detecting the amount of congestion occurring. Here each VM reports its latencies to ResEx and on an average uses 10μs for reporting. This value for reporting is included with the latency to highlight the asynchronous nature of IB communication which may be useful in hiding latencies.

5.7.3 No Interference Handling

We also run two more cases where the interference between the 2 VMs is either negligible or the same as the reporting VM itself. Figure 27 shows the impact of those cases when either of the Reso management algorithms are used. In the figure, FM refers to the FreeMarket algorithm and IOS refers to the IOShares algorithm.

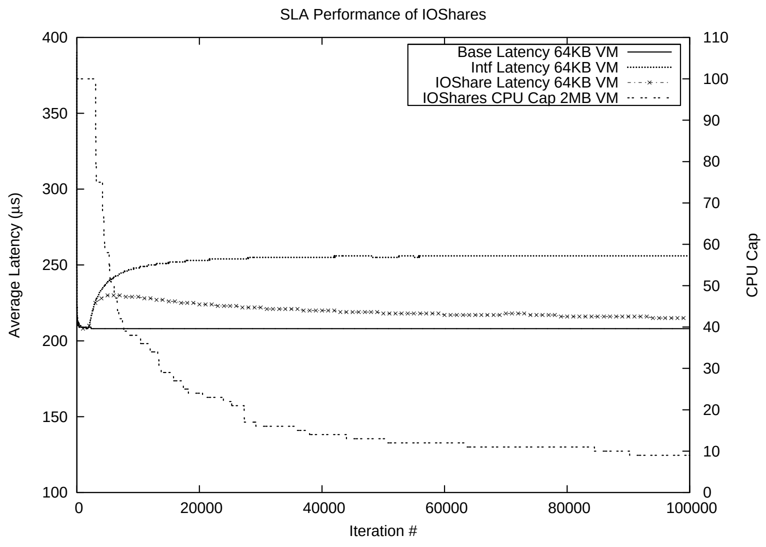


Figure 26: Comparison of application latency when using IOShares algorithm.

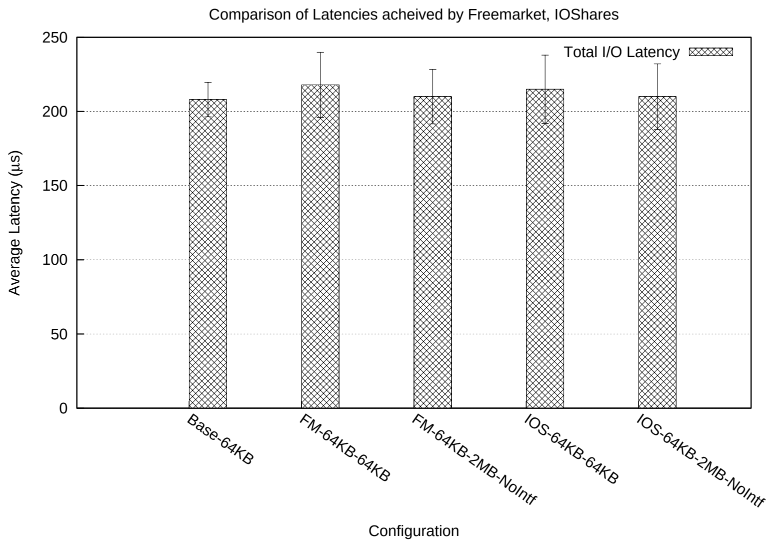


Figure 27: Performance of FreeMarket and IOShares methods on non-interference cases.

The two cases chosen here are, one, when along with the Reporting 64KB VM there is another 64KB VM running Nectere and two, the 2MB VM is issuing requests at

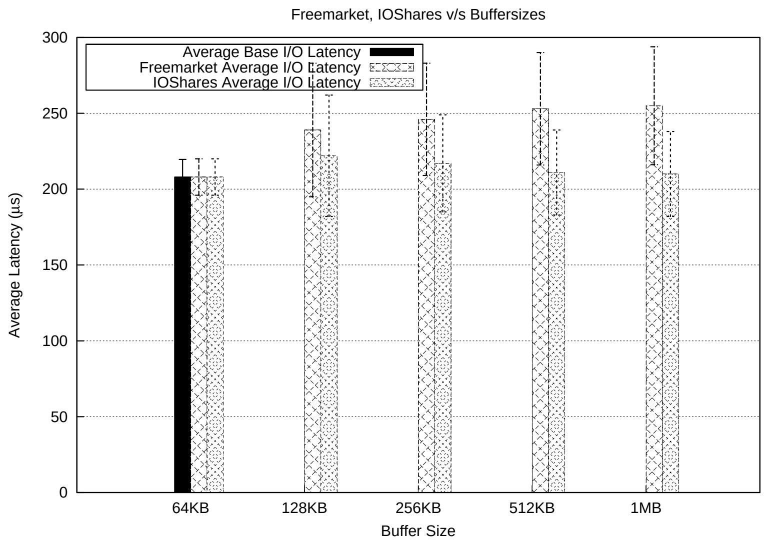


Figure 28: FreeMarket and IOShares behavior with different interfering VM buffer size.

10 requests per epoch (a much slower rate than the interfering VM used in prior experiments).

Figure 27 compares the average latency experienced by the Reporting 64KB VM in these cases with the Base 64KB latency. We see that the values are almost equal to the Base values. This highlights two aspects of ResEx. One, ResEx can not only detect interference for a VM but also back off when there isn't any interference (in case of 64KB-2MB-nointf). Two, ResEx adapts to the I/O performed by the VMs to not penalize VMs if they are doing the same amount of I/O (in case of 64KB-64KB).

5.7.4 Response to Buffer Size

Applications may be configured to use different buffer sizes and therefore ResEx should be able to adapt the different resource management strategies based on that value. Figure 28 shows the latency of the 64KB VM when run against interference VM configured with different buffer sizes. We see that IOShares outperforms FreeMarket by maintaining the average latency very close to the base value.

FreeMarket does not limit the latency since it does not have access to that information. By allowing VMs to 'spend' their Resos as they wish FreeMarket does not eliminate congestion but makes sure the existing resources are used completely. This shows the 'work-conserving' capability of the FreeMarket algorithm. IOShares on the other hand, maintains the running average latency of the VMs and charges VMs causing congestion more. This leads to VMs resources being reduced slowly and as a result the congestion drops. Thus, IOShares is more conservative and aggressive than FreeMarket as a resource management scheme.

5.8 Chapter Summary

In this chapter, we have described ResEx – a resource management approach to improve the ability of virtualized RDMA-based platforms to be shared by collocated latency-sensitive applications. ResEx uses the Virtual Currency Abstraction through the Resos resource unit and mechanisms based on supply-demand economic concepts and congestion pricing to dynamically manage VMs usage of I/O and CPU resources. By using a common "currency" ResEx allows us to establish conversion rates between the two types of resources, which is particularly necessary on RDMA-based platforms which support VMM-bypass, where the hypervisor's only mechanism for dynamically managing the VMs' I/O usage is through appropriate changes in their CPU allocations.

ResEx is implemented for InfiniBand platforms virtualized with the Xen hypervisor, and evaluated using an RDMA-based latency-sensitive benchmark Nectere, modeled after a financial electronic exchange, and for two different resource pricing policies. The results demonstrate that virtualized RDMA-based platforms can be suitable for shared use by latency-sensitive applications. Furthermore, however, the mechanisms and abstractions introduced by ResEx are important for achieving better performance and management of SLA guarantees and for controlling unwanted latency variability.

CHAPTER VI

DISTRIBUTED RESOURCE EXCHANGE: VIRTUALIZED RESOURCE MANAGEMENT FOR SR-IOV INFINIBAND CLUSTERS

In the previous chapter, we demonstrated how we can achieve latency guarantees for collocated latency-sensitive workloads on individual nodes using our Virtual Currency Abstraction. This chapter extends this model to consider distributed workloads deployed on a multi-node and multi-VM setups and their cluster-wide impact on network interference. We first present the challenges that exist in managing these workloads deployed in high performance fabric clusters and thus, motivate the need for distributed, coordinated resource management actions. These actions are implemented via our Distributed Resource Exchange (DRX) model to provide guarantees to latency-sensitive applications collocated with data-intensive distributed applications.

6.1 Challenges to manage distributed workloads

Current data center workloads exhibit diverse resource and performance requirements, ranging from communication- and I/O-intensive applications like parallel High Performance Computing (HPC) tasks, to throughput-sensitive mapreduce-based applications, multi-tier enterprise codes, to latency-sensitive applications like transaction processing, financial trading [126] or Voice over IP (VoIP) services [74].

Furthermore, concerning I/O, commoditization of high-end fabrics such as 40+Gbps InfiniBand and RDMA over Converged Ethernet (RoCE), and hardware improvements for I/O virtualization like Single Root I/O Virtualization (SR-IOV) [77], provide high levels of aggregate I/O capacity and low-overhead I/O operations.

A remaining challenge, however, is the ability to consolidate the above mentioned highly diverse workloads across multiple shared, virtualized platforms. This is because current systems lack the methods for fine-grained I/O provisioning and isolation needed to control potential interference and noise phenomena [85, 106, 114]. Specifically, while SR-IOV-based devices provide low-overhead device access to multiple VMs consolidated on a single platform, this hardware-supported device resource partitioning is insufficient in providing performance isolation. In fact, we demonstrate that for I/O-intensive applications running across consolidated SR-IOV devices, there remain serious issues with performance variability and lack of isolation. We see evidence of this variability in Figure 29 for a transactional latency-sensitive benchmark when it is running by itself versus when consolidated with a throughput-intensive one.

Recent efforts, including our own, have addressed these issues on individual virtualized nodes [26, 39, 44, 112, 79], but effective solutions that span multi-node virtualized infrastructures and distributed, multi-VM applications remain unavailable. This is because there are additional challenges with workloads deployed as distributed VM Ensembles (VMEs), which include:

(i) quantifying the workload characteristics in terms of network I/O guarantees,

(ii) for the timely detection and management of I/O-related interference effects,

(iii) in ways that consider all relevant VME components, and

(iv) take into account all of the physical resources and nodes being used.

This is particularly the case for environments with high-end fabrics, running I/O-intensive workloads, where delays in diagnosing and managing resource congestion and the resulting interference effects, have significant impact on performance degradation [134]. Stated technically, *the effectiveness and timeliness of the performance and isolation management operations concerning the I/O use of distributed VM ensembles requires coordinated resource management actions across the entire set of*

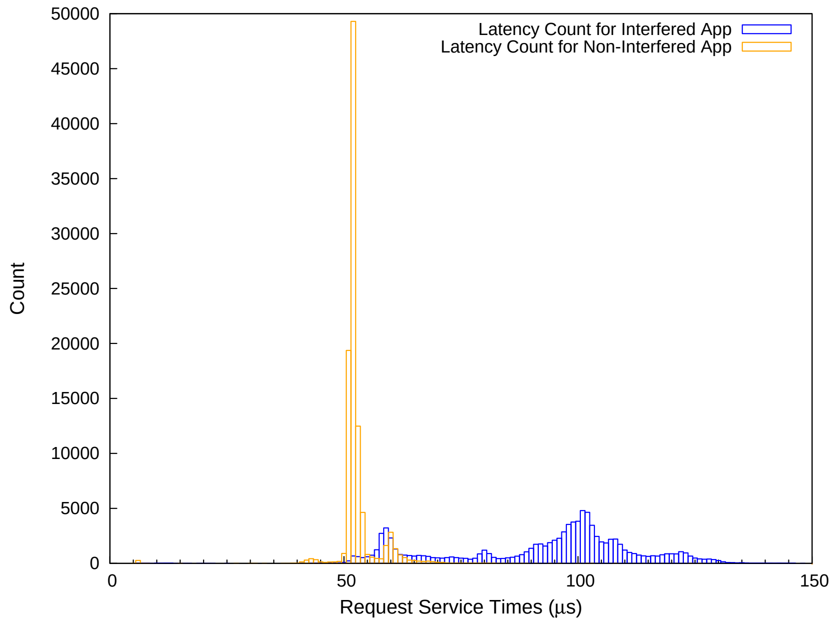


Figure 29: Distribution of Latencies for a Non-Interfered v/s an Interfered Financial Application.

relevant distributed platform resources.

To achieve this goal, we propose a resource management framework called Distributed Resource Exchange or DRX – for managing the performance interference effects seen by distributed workloads deployed in shared virtualized environments. DRX incorporates an I/O workload model to find the resource allocations of workloads in order to satisfy SLAs for a possible set of workloads. DRX enforces these resource allocations by borrowing ideas from microeconomics principles on managing commodities' supply and demand, thereby treating virtualized clusters as an *exchange*. DRX controls resource allocations through continuous accounting, charging, and dynamic price adjustment methods. The price acts a single indicator of interference and therefore, resource sensitivity, allowing us then to translate that value to a realizable resource allocation at the platform level. DRX provides its own methods to interact with the physical platform through a series of platform-dependent resource knobs to ensure that VMEs receive their share of resources. Both the pricing mechanisms and platform-based controls allow DRX to construct a varied range of resource management policies enforceable on such clusters.

Although the ideas used in the DRX design are general, the distributed, coordinated resource allocation actions it enables are particularly important for virtualized clusters with high-end fabrics. Here, the bandwidth and latency properties of the interconnect make them suitable platforms for shared deployment of both I/O- and communication-intensive workloads, and where, precisely because of the I/O-sensitive nature of some of these distributed workloads, the need for low-overhead, effective management actions is more pronounced.

The specific contributions made by this research include the following.

(1) We present the design of the DRX framework and its implementation for cluster servers interconnected with InfiniBand SR-IOV fabrics, and virtualized with the KVM hypervisor.

(2) We develop an I/O workload model which presents a generalized method for SLA satisfaction and resource allocations as input to DRX.

(3) We devise a relationship called the Distributed Causal Congestion Relationship (DCCR) that tracks interference-causing distributed VMEs for a given VM.

(4) DRX integrates mechanisms for low-overhead accounting of resource usage, usage-based charging, and dynamic resource price adjustment.

(5) DRX can be configured to use a generalized method or take advantage of certain platform features to help control the I/O performed by VMs in order to reduce inter-ensemble interference.

(6) The importance of these mechanisms and use of platform features is illustrated through the implementation of two classes of policies:

(i) Equal-Blame Policies - which charges every VM/VME responsible for causing interference equally.

(ii) Hurt-Based Policies - which charges every VM/VME proportionally to the amount of I/O generated by them.

(7) The realization of DRX leverages our own prior work, which developed memory-introspection-based techniques for light-weight accounting, i.e., monitoring of the use of I/O resources in InfiniBand-(and similar) connected platforms [110], and mechanisms for managing performance interference on single node platforms through the use of appropriate charging methods [112].

(8) Evaluations use representative application benchmarks corresponding to transactional, data-analytics, and parallel workloads. The results indicate the importance and efficacy of our distributed management solution and make consolidation on SR-IOV fabrics more feasible for SLA-driven workloads.

The remainder of this chapter is organized as follows. In Section 6.2, we introduce and explain the design of Distributed Resource Exchange (DRX). Section 6.3 describes our the theoretical underpinnings of the mechanisms used by DRX to reason about and manage SLA requirements of the collocated applications and their VMEs. We next describe the DRX mechanisms and their interactions in Section 6.4. In Section 6.5 we provide two examples of dynamic, fine-grained I/O controls that can be used by DRX, a purely software-based one, and another one that leverages available hardware features. This is followed by illustration of two different classes of management policies. Section 6.7 explains our reasoning behind the pricing model used by DRX as well as applicability of DRX to other types of networks. Section 6.8 describes our experimental methodology and measurement results.

6.2 Overview of the Distributed Resource Exchange

Figure 30 shows the architecture of our DRX system with various interactions between the components. At its core, DRX is a multi-level software structure spanning machines and the VM ensembles using them. The main components of DRX are

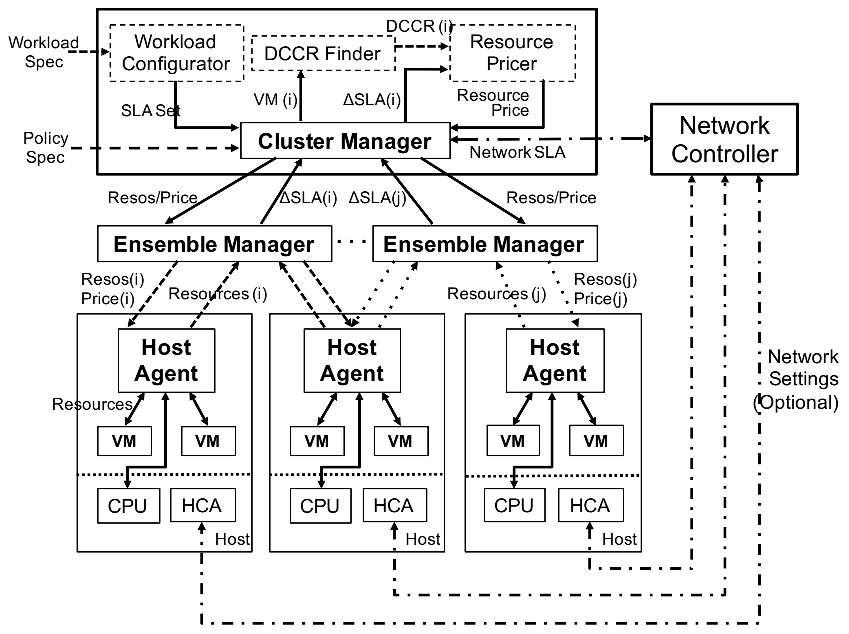


Figure 30: Distributed Resource Exchange Model. This shows the interaction between the various components within DRX. The Workload Configurator, DCCR Finder, Resource Pricer are part of the Cluster Manager. The Network Controller is an entity specific to the type of network (InfiniBand) and the PM can interact with it separately to enforce network guarantees.

the *Cluster Manager (PM)*, the *Ensemble Manager (EM)* and the *Host Agent (HA)*. The PM internally consists of various sub-components like the Workload Configurator, Distributed Causal Congestion Relationship (DCCR) Finder and the Resource Pricer and work with the rest of the PM to enforce cluster-wide policies for providing resource allocations to *VM Ensembles (VMEs)*. VMEs are identified as a set of VMs corresponding to a single cloud tenant or a single application, for example, a multi-tier enterprise workload or a distributed Message Passing Interface (MPI) or MapReduce job.

DRX uses the notion of VMEs as a schedulable entity on the cluster. As with scheduling, each VME is also identified with certain minimum and/or upper-bound guarantees called Service Level Agreements (SLAs). These are provided to the *'Workload Configurator'* as a 'Workload Specification'. The job of the Workload Configurator is to realize the set of VMEs – based on their guarantees, priorities as well as network capacity – that must be satisfied in order to meet a cluster-wide utility function. These selected workloads are then given to the PM in order to correctly enforce the selected policy, which could be as simple as making sure the SLAs for this set are always satisfied. We describe the Workload Configurator in Section 6.3.1.

Since we use VMEs as a schedulable entity, to enforce the selected policy, we have to make resource allocation choices that affect entire VMEs. This also means that there will be interaction between VMEs due to shared platform resources for its constituent VMs. Thus, DRX must also take into consideration this inter-VME interference when making resource allocation choices. This potential interference is found by the *'DCCR Finder'* which defines a relationship between a VM and the constituent VMs of other VMEs, based on the network links being shared. Further, this implies that a single resource allocation for a VME from the PM would translate to many actions – through the EM and the HAs – performed on that VM and/or on the VMs in the DCCR. The DCCR is defined and described in detail in Section 6.3.2.

In the middle tier of our DRX hierarchy are the EMs that are responsible for monitoring the VME's resource and performance needs as well as reporting SLA violations to the PM. They interpret a resource allocation action that the PM issues and interact with the HAs deployed on every host to drive the necessary outcome of the action. The HAs (one per host) maintain accounting information for the host's VM's resource usage, and manage (reduce or increase) resource allocations based on the action from the PM relayed through the EM to them.

In order to facilitate the coordinated resource management actions between various components, the resource management provided by DRX behaves like an *exchange*. We allocate credits called *Resos* to the participants (VMEs and their VMs), and are used to determine the resource allocation for the VMs by the corresponding Host Agent. This allows us to decouple the dependencies between various platform resources and use a single metric of 'Resource Price' to enforce a collective action across VMs. This collective action is enforced through the application of various DRX mechanisms for *allocation*, *accounting*, and *charging*. These mechanisms are performed by each HA in order to enforce a given resource allocation policy. When DRX detects I/O congestion and the resulting performance degradation, it manages the resource by dynamically *pricing* it for a particular VME (and therefore its VMs) that may be causing the congestion as indicated by the DCCR list. The price is determined by the PM – in response to performance interference events – through the 'Resource Pricer' sub-component considering various factors like current price, number of VMs and VMEs, amount of I/O usage. We describe these mechanisms more in detail in Section 6.4.

Intrinsically, DRX contains the ability to map the price changes to actual resource allocations at the platform-level. The HAs are mainly responsible for performing this conversion and it depends on the type of resource, the new price calculated by the PM, the performance degradation experienced as well as the available platform controls. With DRX we focus on data-intensive applications running on multicore servers supporting SR-IOV InfiniBand cards with an abundance of CPU resources. This allows HAs to focus on controlling the VMs' I/O accesses by charging for their I/O usage. We obtain I/O usage from IBMon [110], which samples each VM's I/O queues to estimate its current I/O demand. Controlling I/O usage, however, is not easily done in SR-IOV environments: (1) the VM-device interactions bypass the hypervisor and prevent its direct intervention; and (2) SR-IOV IB devices carry out I/O via asynchronous DMA operations directly to/from application memory. Software that 'wraps' device calls with additional controls would negate the low-overhead SR-IOV bypass solution. The HA, therefore relies on two methods to control I/O usage:
(1) the relatively crude method of CPU capping to indirectly control the I/O allocations available to the VM which have been prototyped before in [112]; and
(2) manipulation of hardware-level congestion settings that control the amount of

data injected by a particular IB Queue Pair and/or application.

In order to use the hardware controls in IB, there are various network settings that must be written in the Network Controller configuration. For IB, it is the Subnet Manager (SM) that enforces the hardware controls by sending specific packets to the HCA after we set the hardware values in its configuration. While these settings seem very specific to SR-IOV IB, the methods in DRX are generic enough to be extended to other networks, for example Ethernet, which do not have such hardware capabilities. The HA can easily be updated to support resource allocations for Ethernet fabrics by updating the resource conversion function. We give more specifics in Section 6.7.

Finally, to deal with dynamism in the workload requirements DRX uses an epoch-based approach: Resos allocations are determined and renewed at the start of an epoch, based on overall supply and demand, per-VM or VME accounting information, etc.; the resource price, along with the charging function, determine the rate at which the workload (some VM or component) will be allowed to consume I/O resources. These mechanisms allow us to treat physical resources like commodities which can then be bought or sold from an 'exchange'. Further, we can use economics-based schemes to control the commodity supply and demand, thereby affecting resource utilization, and the consequent VM performance and interference effects. In the next few sections we describe in detail each of the components of DRX.

6.3 Workload SLA Satisfaction

Given a set of workloads that can be deployed on the shared cluster along with their SLAs, DRX finds the set that can be satisfied as well as which workloads (VMEs/VMs) to charge when a SLA violation occurs after deployment. We explore both these points in this section through our 'Workload Configurator' and 'DCCR Finder' respectively.

6.3.1 Finding Satisfiable SLAs with the Workload Configurator

The Workload Configurator or WC shown in Figure 30 takes an input of the workload specification that defines the SLA (an upper and lower bound in terms of I/O requirements) as well as a priority (weight) for the workload. The priority and SLA of the workload can be negotiated by the tenant with the cluster administrator and the exact process is beyond the scope of this work. We assume a workload model, where each VM Ensemble (VME_i) or Workload (W_i) consists of one or more VMs, identified by its SLA and priority, distributed across nodes in the cluster and is provided to DRX in terms of the specification. Given that a cluster has limited capacity in terms of I/O, there can be instances where not all workloads can have their SLA satisfied at the same time. Thus, the overall goal of WC is to select workloads (or SLAs), based on the priorities assigned subject to the I/O capacity of the cluster.

6.3.1.1 Conditions for selecting Satisfiable SLA

In this work, we are concerned with data center settings where the I/O interconnect is the key source of contention (i.e., VMs' vCPUs are allocated to their own physical CPUs), and therefore use I/O as a key performance metric: (1) Ws' SLAs are described in terms of their I/O requirements - *g*, and (2) Ws' performance is monitored through monitoring their I/O usage at time t, given as *m(W, t)*. Therefore, given a number of workloads ($W_1..W_n$), the goal of WC, stated more specifically, is then

(i) to maximize the weighted sum of the high priority workloads, W_i whose SLA requirements could be met, i.e., $\alpha_i * g(W_i)$, while,

(ii) minimizing the SLA violation or error, e, experienced by the remaining workloads W_j $(e_j(t) = (m(W_j, t) - IOmin(W_j)) > 0)$. This implies that the performance for some workloads must be at the lower bounds of their SLA, where the lower bound is identified by $IOmin(W_j)$.

As a result of the max-min SLA conditions, the workloads are divided into two distinct sets called the *Set-Satisfiable (SS)* and *Set-Victims (SV)*, respectively. We use a Utility Function, U_s to decide which workloads can have their SLAs satisfied given their current performance guarantee, $g(W_i)$ by maximizing the weighted (α_i) sum of the utilities.

$$Maximize \quad \sum_{i=1}^{k} U_s(g(W_i), \alpha_i) \tag{1}$$

subject to,

$$\sum_{i \in SS} g(W_i) < \sum_{n \in N} L_n - \sum_{j \in SV} IOmin(W_j) \tag{2}$$

In Equation 1, the utility function (i.e., weighted sum) can be specified based on number of factors, which include the cluster-wide profit expected from satisfying certain SLAs, the total number of satisfied SLAs, the ability to decrease the aggregate load on the cluster, etc. Equation 2 permits the workloads in set SS to meet their SLAs provided there is sufficient capacity in the links (L_n) for the SS workloads I/O as well as for I/O from all workloads in SV if reduced down to their lower bound or minimum guarantee $IOmin(W_j)$. The minimum value ensures that workloads in SV can continue to make progress and are not stalled indefinitely for lack of network resources. Note, that the WC runs only when workloads are to be admitted into the cluster, to re-balance the SS and SV sets if required.

6.3.1.2 Optimizing Selection of SLA Satisfiable Workloads

The above equations essentially are an example of *Combinatorial Optimization* where the cloud provider wants to optimize their profits or utility when multiple applications are sharing the data center. Generally, Combinatorial Optimizations are illustrated by the 'Knapsack Problem' [30, 103], where we can only fit 'x' items each weighing a certain amount 'w_i' into a 'Knapsack' capable of carrying a certain maximum weight, 'W'. Here, the cloud provider needs to select a certain number of applications to satisfy their SLAs given their guarantees ('$g(W_i)$') and network link capacity ('L_n').

In general, the Knapsack Problem belongs to the NP-Hard family, i.e., it is unlikely that we would find an algorithm that runs in polynomial time. However, we can find approximate solutions using various programming techniques described in [53, 120]. Using the dynamic programming technique to solve the 0-1 Knapsack problem, where each application/VME represents a separate item/instance, we can solve the utility function such that we can find the list of applications that are part of the Set-Satisfiable.

In this work, we use a simple priority-based function to select satisfiable SLAs to simplify the computation required for selecting the workloads. Algorithm 4 shows an approximate solution to finding the SS and SV VME sets for the utility function. The input to the algorithm is the set of VMEs or workloads identified as High Priority (HP) or Low Priority (LP). Next, we place all the VMEs identified as low priority into SV and all VMEs identified as high priority into SS. We sort the workloads in SS by their ascending order of guarantees. Here, we intrinsically assume that workloads with the highest guarantees have the highest utility. Next, we keep inspecting the value of Equation 2 to make sure we are within the limits. If not we move a workload from SS to the SV set. Since we have sorted SS based on ascending order at every step we select the workloads with maximum utility therefore we would still reach our maximization goal as defined by the utility function. The worst case of this algorithm depends on the sorting technique used during *Sort_SS* which is approximately O(n log n) with Merge Sort.

6.3.2 Finding Distributed Interference using DCCR Finder

While the Utility Function provides a way to select the VM Ensembles or workloads which need to have their SLA met, we need to track and identify VMEs that share physical links. This is important since these VMEs can potentially cause interference to one another due to their actions on the same link. The 'DCCR Finder' tracks this *distributed interference* by identifying such VMEs distributed across the cluster.

Algorithm 4 Algorithm for Utility Function Solution

```
Input: WorkloadSpec $W_i = (g(W_i), IOmin(W_i), \alpha_i)$
Output: Set SS, Set SV
 1: Initialize $SV \leftarrow$ Low Priority Apps
 2: Initialize $SS \leftarrow$ High Priority Apps
    /* Store total of SS Workload Guarantees */
 3: $G(W) \leftarrow \sum_{i \in SS} g(W_i)$
    /* Store total link capacity */
 4: $L \leftarrow \sum_{n \in N} L_n$
    /* Store total of SV IOmin */
 5: $IO(W) \leftarrow \sum_{j \in SV} IOmin(W_j)$
 6: Sort SS in ascending order of guarantees
 7: for all Workload $W_i \in SS$ do
    /* Move $W_i$ to SV if $g(W_i)$ unsatisfiable */
 8:     if $(G(W) > (L - IO(W))$ then
    /* Remove $W_i$ from SS */
 9:         $SS \leftarrow SS - W_i$
    /* $W_i$ from SS to SV */
10:         $SV \leftarrow SV + W_i$
    /* Remove $g(W_i)$ from total guarantee */
11:         $G(W) \leftarrow G(W) - g(W_i)$
12:     else
13:         return Set SS, Set SV
14:     end if
15: end for
```

This relationship between VMEs is formally called the *Distributed Causal Congestion Relationship* or DCCR, and is expressed as the set of VMEs potentially affecting the performance reduction of a VM$_i$:

$$DCCR(i) = \{VME_j\}\Big| \; [\text{VM}_k \in \text{VME}_j \wedge \text{VM}_k \in \text{P}(\text{VM}_i)] \tag{3}$$

The DCCR Set is computed by the PM on a per-VME or per-VM level. We choose to do it on a per-VM level since DRX tracks and charges per VM and the Ensemble Manager (EM) reports to the PM about a specific VM(s) not receiving its (their) SLA(s), This is more fine grained than reporting the entire VME as not receiving the SLA. The DCCR Set includes VMEs that share physical links with the this VM, i.e. VMEs with VMs that are on the same physical machine as VM_i, denoted by the function P.

After computing this set we identify those VMs within these VMEs that are actually causing the interference and infer the amount of interference by using the I/O monitoring data. The resulting VMEs can also be present in the SS Set selected by the utility function, in that case, we can only limit their resource allocations, and corresponding performance degradation, down to their SLA value. The interference amount, i.e. the performance degradation, is then used in our 'Economics'- and 'Control Theory'-based DRX mechanisms which compute resource allocations for these VMEs. We discuss these mechanisms along with their use in specific SLA policies next.

6.4 DRX Resource Management Mechanisms

The overall goal of DRX is to provide SLA guarantees to certain applications identified by the utility equation, and it must do so for dynamic workloads without any pre-existing knowledge of the workload characteristics. The DRX platform, therefore, implements a range of mechanisms which work together in order to satisfy the workloads selected by the Workload Configurator. These mechanisms are centered around three aspects: (a) Use of our 'Virtual Currency Abstraction' that uses 'Resos' as a Resource currency for VMs which enables Exchange transactions for EMs, (b) Supply-Demand principles that govern the distribution and pricing of resources to VMs, and (c) Control theory semantics to understand the change in performance of VMs when their resource allocations change. The 'Resource Pricer' sub-component of the PM is responsible for enforcing these three aspects and relies on the input from the DCCR Finder about the set of interfering VMEs as well as the performance degradation value reported by the EMs. In this section we explain the implementation of the mechanisms and how the Resource Pricer uses each of these aspects.

6.4.1 Allocating, Accounting for Resos and Charging Resources

The PM is responsible for global resource management of the cluster and it ensures resources are allocated in an appropriate manner to meet the resources of the VMs. However, to improve scalability for VM resource management, the PM allocates a certain number of Resos per EM called 'EM Allocation'. Each EM Allocation depends on the set of all resources present in the cluster, i.e., 'Resource Supply' and on the resource management policy, see Section 6.6. Further, each EM is responsible for distributing Resos to its VMs. For the sake of simplicity, we assume EMs distribute Resos to its VMs 'equally', unless we state otherwise is our policies. Since we are focused on IB resources for management, we assign Resos to VMs only for these resources.

In order to deal with resource dynamism in the cluster, we use an 'Epoch and Interval-based model' for accounting of resources. We distribute Resource usage credits – Resos – at the start of each epoch. Resource accounting is performed at specific time intervals, where multiple time intervals constitute an epoch. Specifically, every interval, the Host Agent deducts Resos from the VM's Resos allocation – i.e., charges VMs – to account for the I/O consumed by each VM in that interval. We use an epoch of 60 seconds and an interval length of 1 second. These values allow a suitable balance between the overhead and accuracy of accounting and resource allocation for VMs. We next describe how we use our 'Virtual Currency Abstraction' for resource pricing within our DRX model.

6.4.2 Resource Pricing using Virtual Currency Abstraction

First, the price increase intrinsically depends on the amount of congestion-caused *Performance Degradation (PD)* of a VM/VME. We find the PD for a VM using IBMon to detect changes in the IB usage. The PD is the percentage change in the CQEs (for RDMA) or I/O Bytes (for IPoIB or RDMA port counters) generated by the VM. To maintain a certain SLA the VM needs to generate a required number of CQEs or Bytes. When this CQE rate falls below the SLA, IBMon detects it and we can report the difference between it and the SLA to the HA. The HA further reports this to the EM of that VM which in turn sends the value to the PM. This SLA value in CQEs can be pre-computed based on the I/O requirements of the workload, before the workload is deployed on the cluster. However, the exact computation is beyond the scope of this work and we assume that it is provided to us as a workload input [99].

Second, pricing is performed on a *per-VME Basis*. This simplifies our ability to track prices across the cluster when pricing for an entire VME and reduces the amount of communication performed between HAs and the PM. By performing price changes on the entire VME, we can provide a faster response to reduce congestion, rather than repeatedly changing prices per VM. All VMEs that belong to the DCCR set of a VM whose performance degradation triggers a price adjustment, will have their price increased. The amount of the price increase for each VME would depend on these factors listed above, as well as on the cluster-wide policy.

Third, in order to be more flexible in changing the price based on the policy we use two parameters α_i and $\delta\mathrm{P}_i$, which control the price increases for a VME_i. α_i defines the weight or priority for the VME_i and is between 0 and 1 as defined in the Workload Specification. $\delta\mathrm{P}_i$ is the policy coefficient for a VME_i and is defined for each policy in Section 6.6.

Fourth, pricing is performed in a *continuous manner*, i.e., the next price is based on the previous price and is updated based on observed performance degradation. This allows us to use Control Theory to adapt the change in price to the dynamism of the workloads running in the cluster. This is important since we do not know apriori what is the specific impact on price changes per workload. This price change will also affect the underlying physical resource allocation for the affected VMEs. We therefore utilize a Proportional Adaptive Controller, similar to controllers described in [102, 135], to realize our Pricing function for a VME and is shown as follows.

$$\begin{aligned} p_i(t) &= f(p_i(t-1), e_j(t), \delta P_i, \alpha_i) \\ p_i(t) &= p_i(t-1) + K_i(t) * e_i(t) \\ K_i(t) &= \delta P_i * \alpha_i, \ \ 0 < K_i(t) < 1 \end{aligned} \tag{4}$$

In Equation 4 the terms K refers to the 'gain' of the respective control functions. It is defined to be dependent on the performance degradation experienced by the workloads and converts this error into Price changes for congestion causing workload. The δP_i and α_i parameters affect how sharp the price increase for a particular workload should be, and are further discussed in the context of specific policies in Section 6.6.

6.5 Platform Controls for RDMA I/O

While the price of the resource computed for a VME identifies the amount of I/O interference caused, we need to convert that value into realizable resource allocations without apriori workload knowledge. The resource allocation value depends on the effectiveness of the control on the I/O performed as well as how flexible the control is in setting the value. We discuss these factors and their applicability for two specific platform controls - CPU Capping and InfiniBand Hardware Features - for controlling RDMA InfiniBand I/O.

6.5.1 Software-Based Controls - CPU Capping

We can use the CPU Scheduler on each host to change the amount of CPU, i.e. CPU Cap for each VM. As a result the VM gets lesser time to send a RDMA request which goes directly to the hardware, thereby giving us a crude method to control the I/O the VM performs as shown before in [112, 111]. CPU Capping is a host-level control, that can be assigned per VM and has a unit-level granularity. While it does not translate to an exact resource allocation, the impact on the I/O is easily evident and configurable to provide the necessary reduction in I/O interference.

The capping degree depends on the policy being implemented. In general, the CPUCap for a VM depends on the New Price for the VME_i ($p_i(t)$), Old Cap for VM ($c_i^j(t-1)$), VME Priority (α_i) and a CPUCap policy coefficient, $\delta \mathrm{C}_i^j(t)$, which defines the conversion of Price into a CPU Cap. Here we again use a Proportional Adaptive Controller which takes into account the previous Price change and CPU allocated to the VM to compute the new CPU.

$$\begin{aligned} c_i^j(t) &= f(c_i^j(t-1), p_i(t), \alpha_i, \delta C_i^j(t)) \\ c_i^j(t) &= c_i^j(t-1) - Z_i^j(t) * p_i(t) \\ Z_i^j(t) &= \delta C_i^j(t), \ \ 0 < Z_i^j(t) < 1 \end{aligned} \tag{5}$$

In Equation 5, the term Z refer to the 'gain' of the control function. The gain Z, defines the amount of CPU to be deducted based on the New Price computed. Z is defined such that it is workload specific rather than policy-dependent, by capturing the workload 'I/Oness', i.e., the change of I/O with CPU, which is different for different workloads.

6.5.2 Hardware-Based Controls - InfiniBand Hardware Features

Here, we describe the use of features supported on newer generation InfiniBand adapters for DRX resource management. These features include Congestion Control and Quality of Service support and can provide differing performance characteristics to applications based on IB Service Level/Queue Pair used. These features are set in the configuration of the Network Controller, i.e. the Subnet Manager and are designed to service the IB Queue Pairs based on the values set and the feedback on congestion from the network devices.

While the hardware features provide a low-overhead mechanism to perform resource management for RDMA-based fabrics, one of the limitations of using these features is the flexibility/granularity in setting the values for enforcing resource management. The limitation comes from two factors: One, that the values apply to the entire HCA and not individual VMs so in order to provide guarantees for a VME, each VME must be identified by a SL. Two, the features are designed to be work-conserving such that there is no unused bandwidth maintained, which works against DRX to reduce actual I/O performed by the VM.

6.5.2.1 Congestion Control

InfiniBand networks use a credit-based mechanism for flow control where ports are assigned a certain number of credits for sending packets. The port can continue to send packets as long as it has credits available. The credits are replenished at intervals. It is possible that when multiple ports send data into the network certain links are overused. This prevents ports from injecting data even though it may have credits available. Such a state for a port is called 'Congestion' and can be mitigated by utilizing the 'Congestion Control' mechanism on InfiniBand hardware [117, 144].

This mechanism allows setting parameters on hardware that control the rate of injecting packets by QPs on the HCAs. The InfiniBand specification [52] describes these parameters for both HCAs and Switches as well as the values to be assigned to them. Switches use the Forward Explicit Congestion Notification or FECN bit in the IB packets to inform destination HCAs that there is congestion present in the link. The destination HCA will send a Congestion Notification Packet or CNP with the Backward Explicit Congestion Notification or BECN bit set to the source HCA indicating that the QP should reduce its send rate. The parameters described in the specification govern the rate of sending BECNs, injection rate delays for QPs/SLs, and logging congestion information.

The parameters are divided into 2 groups, one called 'Congestion Control Table' which provides a list of injection rate delays (IRD) to be used, two called 'Congestion Control Parameters' which denotes how the preceding table is traversed. The following parameters are the most relevant for enabling congestion control:

- Congestion Control Table Index Minimum (CCTIMin): The minimum index to be used in the CCT when congestion occurs.
- Congestion Control Table Limit (CCTILimit): The maximum index to be used in the CCT for congestion.
- Congestion Control Table Index Increase (CCTI_Inc): The increase in the CCT Index when a BECN is received on that HCA.
- Congestion Control Table Index Timer (CCTI_Timer): After this timer value expires the CCTI is reduced by 1. The timer is reset if a BECN is received for that QP/SL.
- Congestion Control Table Shift (CCT_Shift) and Multiplier (CCT_Mult): Both these values together specify how the IRD changes as BECNs are received by the HCA. These are 2-bit and 14-bit values respectively and the maximum IRD possible is $\approx$3.4ms.

We set these values for the HCA nodes by sending InfiniBand MADs (Management Datagrams) to the Subnet Manager (SM). The SM then forwards these values to the appropriate HCAs using MADs. With DRX the PM can send these MADs to the SM when HAs report a SLA violation. The PM can continuously change the value of the IRDs for certain QPs, i.e., the CCT_Shift and CCT_Mult values in order to reduce congestion from these QPs. We configure the HCAs to apply congestion control on a port Service Level (SL) basis and set the 'ControlMap' in the SM accordingly. As

BECNs are received the CCTI will be incremented for that SL and hence any QPs using that SL will be serviced after a delay as specified by the CCTI.

6.5.2.2 Quality of Service or QoS

An InfiniBand link is divided into several Virtual Lanes or VLs, each of which can be assigned to a different application. Each VL can be configured to provide differentiated service to the application attached to it. Here, each application is assigned a Service Level (SL) for the QP associated with that application. The QoS feature of InfiniBand configures the mapping of these SLs to VLs called 'SL2VLMapping'. The HCA performs arbitration of VLs based on a table called the 'VLArbitration' table containing weights for each VL. The weights govern the proportion of the bandwidth allocated to that VL. Each unit of weight allows the HCA to service that VL for one chunk of 64 bytes. The VLs are also divided as low and high priority VLs. High priority VLs are serviced up to the 'High Limit' value in order to prevent low priority VLs from starvation. By configuring higher weights to certain VLs and mapping SLs to these VLs, we can ensure higher bandwidths for these VLs. The HCAs ensure processing of VLs in a 'work-conserving' manner, i.e. the HCA will keep the link utilized as long as some VL has data. This nature of QoS is very useful for throughout-oriented applications but affects latency-sensitive applications since they always need their data to be immediately processed.

The QoS mechanism allows specification of a simple policy consisting of mapping Upper Layer Protocols or ULPs, like SDP or IPoIB, to different SLs as well as weights or mapping for these SLs. We use this simple policy injunction with the congestion control mechanism to provide resource management for the applications. We set these QoS values in the configuration file of the Subnet Manager and therefore, these values apply to QPs that were created after the SM is restarted with these values set. Next, we describe how we use the platform controls along with the DRX mechanisms to provide a varied range of policies that can be enforced on such clusters.

6.6 Policies for a Distributed Resource Exchange

Given the various components and mechanisms of DRX in Section 6.2 and 6.4 we now describe various ways in which these components interact using the Workload Model described in Section 6.3 to provide distributed resource management. We divide our policies into two classes based on the interference charges to the VMEs, i.e. whether all VMEs are equally charged or are charged based on the percentage of interference caused. We further categorize policies via the 'Control Mechanism' for reducing interference - Pricing versus purely using Platform Controls. The policies constructed using only 'Platform Controls' eschew the Pricing mechanism and directly apply resource allocation values based on some policy parameters. This allows us to compare the effects of whether the Price is a good indicator of interference as well as a vehicle for performing distributed resource allocation actions. Table 4 shows the list of policies that we have implemented for DRX. Also, Tables 5 and 6 show the specific configuration values we use the InfiniBand Congestion Control- and Quality of Service-based Platform Control policies, respectively.

Table 4: Various Policies implemented in DRX based on amount of interference.

Control Mechanism	**Equal-Blame**	**Hurt-Based**
Distributed Pricing	EB-Dist	HB-Dist
Local Pricing	EB-Local	HB-Local
Platform-Based	CC-Dist	PC-Dist
	StMax	StMaxQ
	DRXMax	DRXMaxQ

Table 5: InfiniBand Congestion Control Parameter and Values.

IBCC Parameter	Value
CCTI_Timer	32767
CCTI_Inc	10
Trigger	100
CCTI_Min	10
CCTI_Limit	127
CCTI_Shift	0
CCTI_Mult	16383

Table 6: InfiniBand QoS values for various Policies. SL0 for Transactional Workloads, SL1 for other workloads. All Policies have same IBCC Values.

Policy	QoS Values	
	SL-VL	VL Weights
StMax	0,1	32,32
DRXMax	0,1	32,32
StMaxQ	0,1	254, 1
DRXMaxQ	0,1	254, 1

6.6.1 Equal-Blame Policies

In general, this type of policy treats interference from all VMEs 'equally' and therefore, charges them the same amount whether as a Price or directly as a reduction in resource allocation.

6.6.1.1 EB-Dist and EB-Local Policies

We implement these policies to show a naive method of charging VMEs when there is congestion. The difference in the two policies is which VMEs/VMs are charged when congestion occurs. In EB-Dist every VM of a VME present in the DCCR of congested VM x is charged, termed a 'Distributed Policy'. In EB-Local, only VMs present on the same host as the congested VM x have their prices increased, a 'Local' policy. For example, if the performance degradation reported by an HA 'y' is 30%, then with 2 VMEs in the DCCR of congested VM 'x', each VME's price is increased by 15% for the EB-Dist Policy. For EB-Local, only the VMs belonging to the DCCR and present on host y will have their price increased by 15%. The Price and CPU Cap functions for a VME_i and VM j as follows:

$$p_i(t) = p_i(t-1) + (p_i(t-1) * \delta P_i * \alpha_i) * e_j(t) \tag{6}$$

$$\begin{aligned} c_i^j(t) &= c_i^j(t-1) - c_i^j(t-1) * \left(\frac{p_i(t) - p_i(t-1)}{p_i(t-1)} * \delta C_i^j(t) \right) \\ &= c_i^j(t-1) \left(1 - \left(\delta P_i * \alpha_i * e_j(t) * \delta C_i^j(t) \right) \right) \end{aligned} \tag{7}$$

$$\delta P_i = \frac{1}{N(DCCR_x)} \quad \delta C_i^j(t) = \frac{EL_h}{RL_i^j(t)}$$

where, N(DCCR$_x$) denotes the number of VMEs in the DCCR set of VME x, EL$_h$ and RL$_i^j(t)$ denote the % of Epoch Left on HA h and % of Resos Left for VM j $\in$ VME_i respectively.

For this policy the Pricing function is a linear function that increases prices when the error is larger than the required SLA value. The coefficients δP_i and $\delta C_i^j(t)$ were specifically chosen to highlight the equal contribution of the interfering workloads and to penalize VMs for using their Resos at a faster rate than the epoch progression.

6.6.1.2 Platform-Based Policies

CPUCap Distributed (CC-Dist) - The goal of this policy is to only use the software-based platform controls to validate the benefits of Pricing, i.e if we are converting Prices to CPU Caps using CPUCaps directly should help as well. Since all interfering VMs are charged the same, we decrease the CPU Cap steadily by 5% (up to 25%) for all these VMs distributed across the entire cluster.

Static Max (StMax) - We use the Congestion Control feature in IB to control the interference caused. This policy is equivalent to the CC-Dist policy since we are charging all VMs equally for interference by adding the maximum possible value of IRD for their queues. These values are applied directly at the SM without the intervention of DRX and therefore are static. The QoS feature is enabled but equal weights, i.e. equal priorities are assigned to every SL.

DRX Max (DRXMax) - We can leverage DRX's ability to track an application's SLA violation and then specify IBCC values to minimize the violations. Here, we use the same maximum IRD values as in StMax and DRX applies these values when it receives a SLA violation message from the HA. Here, the PM sends Management Datagrams (MADs) to the Subnet Manager in order to apply the values. Also, the QoS mechanism is configured in the same way as StMax.

6.6.2 Hurt-Based Policies

This type of policy uses the ratio of I/O generated by the VMEs to calculate a price increase since the I/O generated, i.e. 'Hurt' would proportionally impact the performance degradation more.

6.6.2.1 HB-Dist and HB-Local Policies

The Distributed and Local policies imply the same meaning from before. From the earlier example, if the VMEs perform I/O in a ratio of 9:1, the effective price increase for VME1 would be 27% and VME2 3% for 30% PD. The PM finds the aggregated I/O from the HAs to compute the I/O ratio between VMEs and therefore adjusts the prices accordingly. The goal of this policy is to charge the VMEs more based on the fraction of performance degradation they caused. For this policy δP_i and $\delta C_i^j(t)$ in Equations 6, 7 for VME_i and VM j are as follows:

$$\delta P_i = \frac{IO_i}{\sum_{k=0}^{N} IO_k}, \quad \delta C_i^j(t) = \frac{EL_h}{RL_i^j(t)}$$

where, N denotes the number of VMEs in the DCCR set of VME x and IO$_i$ the amount of IO performed by VME_i. The rest of the formula is the same as the Equal-Blame Policy.

The δP_i term makes sure the Price increase tracks the amount of I/O generated by the VME_i and is therefore uses a linear adaptive gain in order to provide a better response to minimize SLA error.

6.6.2.2 Platform-Based Policies

Proportional CPUCap (PC-Dist) - In PC-Dist, we decrease the CPU Cap for the interfering VMEs based on their I/O Ratio metrics. Therefore, each VM would have its CPU Cap reduced by a fraction of the 5% based on the I/O Ratio.

Static Max QoS (StMaxQ) - We use the VL prioritization and weight assignment along with the maximum IBCC values to give a higher priority to transactional workloads like Nectere [45] and lower priority to other workloads. Therefore, we simulate a Hurt-Based policy since workloads that send out more I/O would be penalized more, i.e. get less bandwidth.

DRX Max QoS (DRXMaxQ) - This is again similar to the StMaxQ policy but the application of IBCC parameters is performed by DRX rather than set manually. Also, we use the VL weights assignment to prioritize transactional workloads over other applications.

6.7 Discussion

6.7.1 Comments on DRX Pricing and Workload Control

Integral to the Pricing and CPU Cap Equations is the reaction of the workloads to changes in these quantities. DRX tries to capture the relationship between Pricing, CPU capping and Workload I/O changes using these equations. This depends on how closely DRX can 'track' the workloads' response to resource changes and this limits the effectiveness of DRX on such workloads. Therefore, when DRX detects workload I/O changes it can infer whether those changes are within 'limits' which DRX can control. We establish certain conditions on DRX Pricing.

When does DRX limit Price changes?

We limit price changes to one per interval which allows workloads to adapt to the change in price, therefore change in resources. Also, we limit the maximum price that can be charged to the VME Ensemble as $MaxPrice_i = \frac{ResosAlloc_i}{IntsPerEpoch}$ and set the price as the minimum between this $MaxPrice_i$ and $p_i(t)$. This prevents unbounded price increases.

When does DRX set minimum resource allocations?

If we set MaxPrice we also allocate the minimum I/O for that VME, $IOmin(W_i)$ which implies that we set minimum values for the physical resources like CPU for the individual VMs and prevent any further price changes. Conversely, if we allocate $IOmin(W_i)$ for a VME i during the epoch through price changes, then we prevent any further price changes for VME i.

What are the conditions for DRX stability?

Since we use DRX for controlling RDMA-based workloads through CPU Capping, stability of our control functions is important. We define stability as the tendency of DRX to allow SS workloads to meet or exceed their SLA while keeping SV workloads at or above their IOmin values. Therefore, DRX is stable when the price increases for SV correspond to an appropriate decrease in their respective I/O, i.e.
$\frac{p_i(t)-p_i(t-1)}{p_i(t-1)} \approx \frac{m_i(t)-m_i(t-1)}{m_i(t-1)}$

Stated differently, and based on the properties of the control theory model DRX uses [102, 135], there is an explicit relationship between the selected parameters of the pricing function (i.e., α_i, K and δP_i) and the type of workload variability that DRX can successfully manage. Furthermore, as long as this condition is met, DRX will continue to make progress towards the IOmin value for SV workloads as well as make sure the SS workloads can meet their SLAs.

6.7.2 Applicability towards non-RDMA networks

The methods described in this chapter are of a general nature and can be applied for other types of networks as well, for example, Ethernet. Intrinsically, our DRX solution relies on three specific network properties in order to work effectively.

One, the underlying SLA workload model relies on the *accuracy of the SLA* specified and that value may need to be updated when the application is run on a slower bandwidth Ethernet network. This means that the Utility Function algorithm will behave correctly to select the right workloads to be satisfied.

Two, the pricing strategies defined in DRX use the *monitoring data* from IBMon to decide whether a workload should have its priced increased or not. For Ethernet networks the pricing strategies can be updated to take into account Ethernet frames sent/received by the application instead of the IB MTU. The HAs can acquire this information directly by using tools provided by the OS.

Three, the *control knobs* that DRX uses for reducing the amount of I/O an application performs must be changed for Ethernet networks. DRX relies on both software and hardware-based control knobs due to the OS-bypass model of InfiniBand, for Ethernet however, these knobs can be completely in software as described in [122]. Here, the application data can be buffered in the OS/Hypervisor till it is the application's turn to transmit the data. The buffer time is dependent on the application SLA and the current network price imposed on the application.

6.8 Evaluation

6.8.1 DRX Implementation

We have implemented DRX in C and it utilizes the libvirt [76] for interacting with the KVM hypervisor and the ZeroMQ library [143] for distributed communication between the DRX Components (PM, EMs, and HAs). In order to simplify implementation, our HAs also act as EMs for the VMs that are present on that host, i.e.

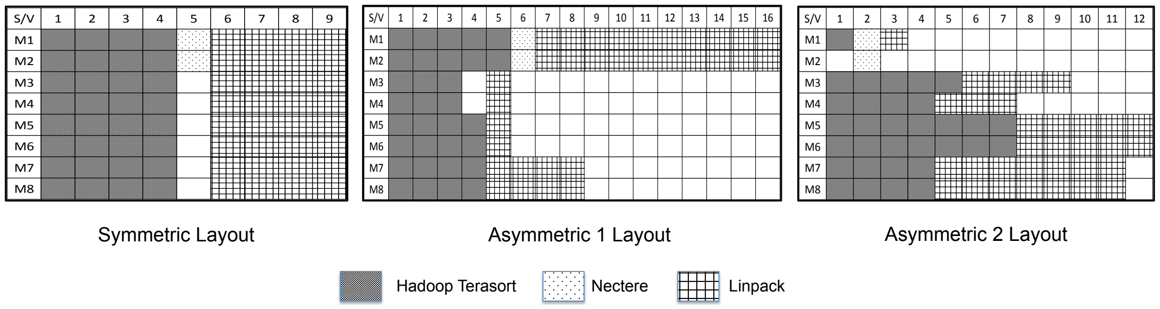


Figure 31: Workload Layouts in Cluster

they react and response to VM's SLA's degradation as an EM and allocate resources as required as an HA. This reduces the amount of communication and computation required between the components. The PM resides on a pre-defined machine possibly with other VMs and an HA, while HAs are assigned one per host. For CPU Capping, we rely on Linux 'Control Groups' [116] that limit the CPU bandwidth for a VM process. We set the 'cpu. cfs_period_us' and 'cpu.cfs_quota_us' parameters to set a CPU Cap. We use the KVM libvirt function 'virDomainMemoryPeek' in IBMon to perform memory introspection of the guest to interpret IB QP and CQE information.

6.8.2 Testbed

Our testbed consists of 8 Relion 1752 Servers. Each server consists of dual hexa-core Intel Westmere X5650 CPUs (HT enabled), 40Gbps Mellanox QDR (MT26428) ConnectX-2 InfiniBand HCA, 1 Gigabit Ethernet and 48GB of RAM. Our host OS is RHEL6.3 OS with KVM and the guest OS' are running RHEL6.1. Each guest is configured with 1 VCPU (pinned to a PCPU), 2GB of RAM and an IB VF. We use the mlx4_core beta version of the drivers (based on OFED 1.5) for the hosts and guests, configured to enable 16VFs, so we can run up to 16VMs on each host. The IB cards are connected via a 36-port Mellanox IS5030 switch.

Table 7: Workload Details. The #VMs column indicates the number of VMs assigned for that benchmark followed by the VMs on every host in brackets.

Workload	**Data Size**	**#VMs**
Nectere	64KB	2 (1)
Hadoop - TeraSort	10GB	32 (4)
Linpack	300 1000 7500 (Ns)	32 (4)

6.8.3 Workloads

We use three benchmarks, each representing a different type of cluster workload. Nectere [45], is a server-client-based financial transactional workload with low latency characteristics. We measure its performance in terms of μs for request completion. We use Hadoop's TeraSort [46] with a 10GB dataset as a representative for data analytic computing to generate distributed interference. For Hadoop workloads we use the job running time as the performance metric. Linpack [49] is a characteristic MPI workload for clusters and uses Gflops as its performance metric. Table 7 summarizes the parameters for our workloads used. We run the workloads in a staggered manner, where we start and let the Hadoop job run for 30s before starting the Linpack job. Next, we start the Nectere workload and run all the jobs till Nectere completes successfully. This ensures that the workloads are performing sufficient I/O communication before Nectere starts.

Figure 31 shows in detail the layouts of the VMs for each of the workloads and layouts described here. Each row in Figure 31 denotes the node and the VMs configured on that node. In the 'Symmetric Layout' each physical machine is running 4 VMs of each benchmark. Additionally, we also use 2 Asymmetric Layouts – *Asymmetric-1* and *Asymmetric-2*. In Asymmetric-1, we have more Linpack and Hadoop VMs (up to 16 total) on the same physical machine as the Nectere VMs. Therefore, this layout should cause more interference to the Nectere application. In Asymmetric-2, there

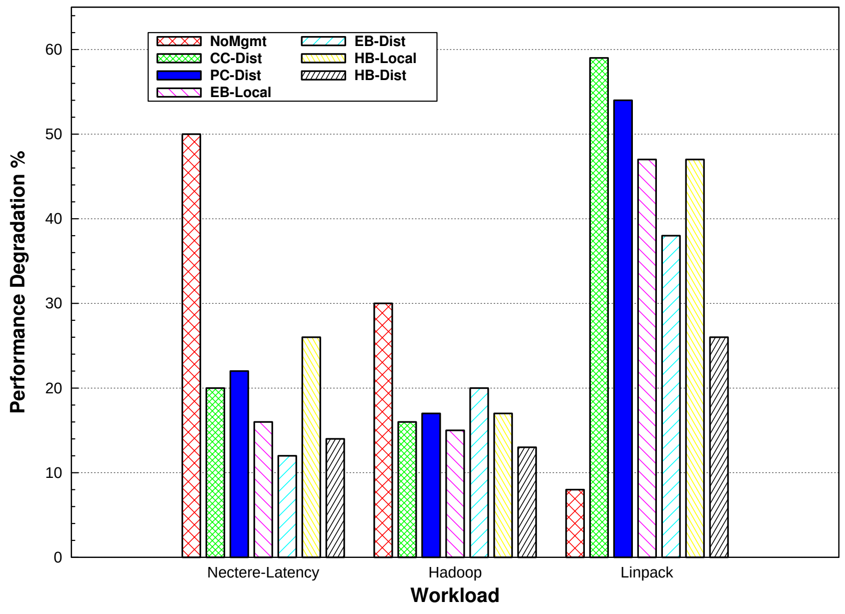


Figure 32: Effect of Policies on Performance of Nectere, Hadoop and Linpack Workloads.

is only one VM each of Linpack and Hadoop along with the Nectere VM. This explores the other end of the asymmetry, where there is minimal interference. These are asymmetric in the sense of how distributed the workload VMs are in the cluster.

In general, we run each policy for the 'Symmetric' layout unless we mention explicitly the type of layout. Each DRX policy is run one at a time and it ensures that when Nectere is running, it maintains the CQE/s within a SLA limit of 15% (we can easily configure other values) from the base value. We design our experiments in order to highlight some of the important features of DRX and their impact on the selected workloads.

Broadly, we divide the results into three different categories: (i) policy performance, (ii) policy sensitivity to resources usage patterns, (iii) limitations and overhead of DRX in different workload layouts, and (iv) effectiveness of hardware features, which we describe next.

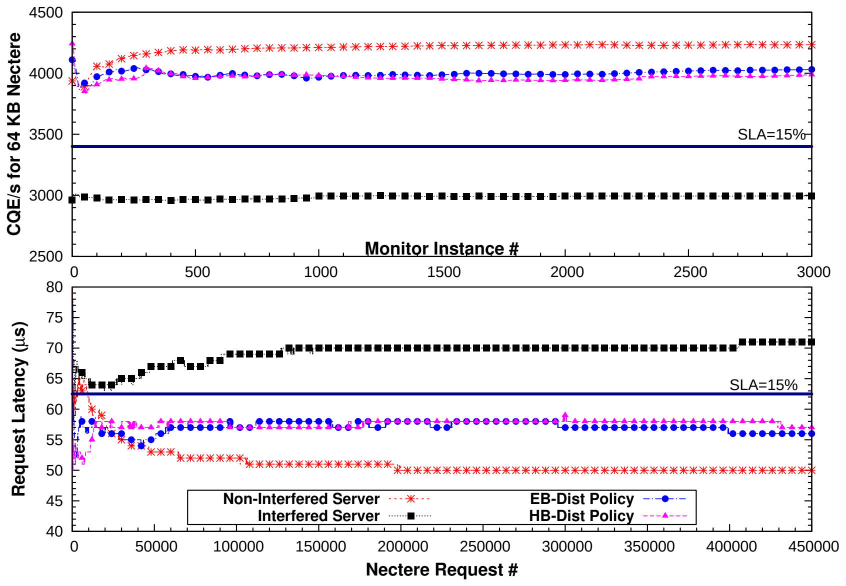


Figure 33: Effect of Policies on the running average of Nectere CQE/s and Latency. High CQE/s denotes Low Latency.

6.8.4 Policy Performance

Figure 32 shows the impact of the distributed and local policies on workload performance. The CC-Dist and PC-Dist policies do help in reducing Nectere latency, but not to its SLA level. These also have a much greater impact on the performance of Linpack and Hadoop, because of the continuous capping. The EB-Local and HB-Local policies cannot reduce the latency for Nectere below the SLA because Linpack VMs on other machines are actually causing congestion by sending data to its VMs collocated with Nectere. However, in the case of the EB-Dist and HB-Dist policies, Nectere can meet its SLA of 15%. This demonstrates *the feasibility of resource pricing as a vehicle to reduce congestion, as well as the importance of performing distributed resource management actions, as enabled by DRX.*

Figure 33 shows the impact of the EB-Dist and HB-Dist policies on the latency of the Nectere application (bottom graph). It also shows the change in the metric used

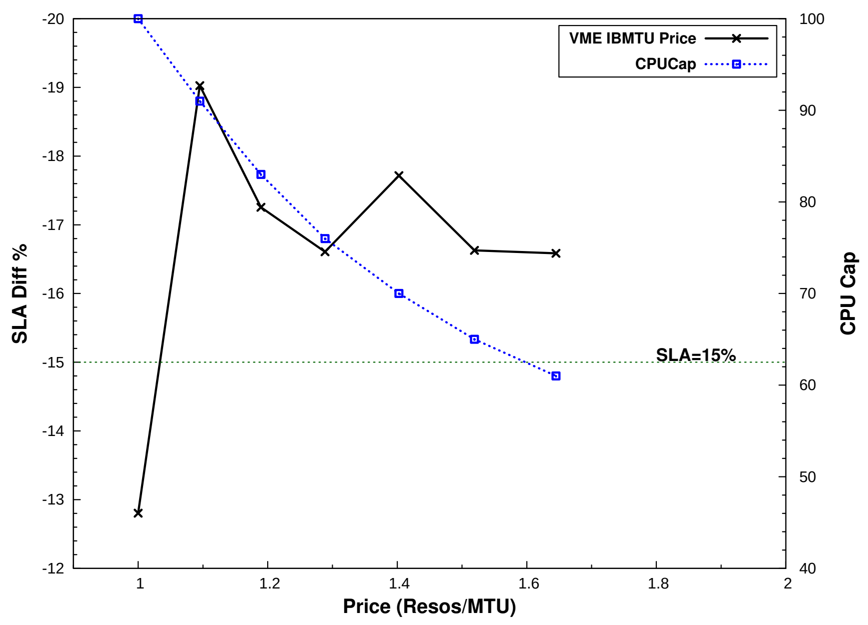


Figure 34: Equal Blame Policy: Comparison of VME Prices, CPU Cap and SLA Difference.

by DRX to manage I/O performance – CQE/s. Both the Equal-Blame and Hurt-Based policies are effective in reducing contention effects and providing performance within the guaranteed SLA levels – they reach the same value for latency, though their impact on the interfering workload performance is different. Also, the HB-Dist policy provides the least degradation of Hadoop and Linpack workloads while meeting the SLA for Nectere.

Figures 34, 35 show the impact of the EB-Dist and HB-Dist policies respectively on VME Prices, CPU Cap and SLA Difference. The CPU Cap here denotes the average CPU received by the VMs belonging to the respective workload. The EB-Dist policy degrades the workloads more since it increases the prices equally for both VMEs which negatively impacts how fast the CPUCap is reduced. As a result the EB-Dist policy is more reactive or *fast-acting* to SLA violations as highlighted by the CPU Cap reductions versus price increases shown in Figure 34. EB penalizes interfering VMs more than HB and assesses a lower CPU Cap for Hadoop, Linpack

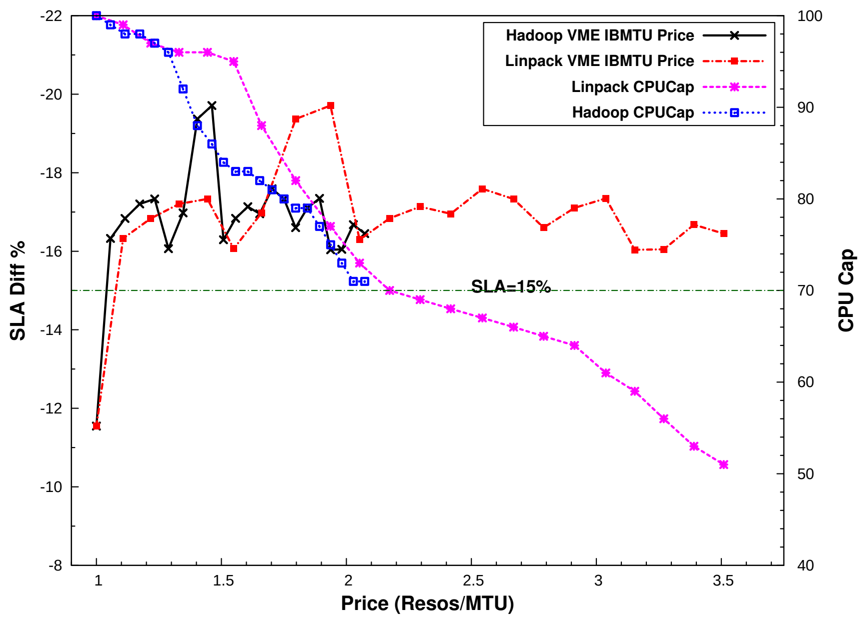


Figure 35: Hurt-Based Policy: Comparison of VME Prices, CPU Cap and SLA Difference.

at 60. Figure 35, shows the more *slow-acting* nature of the HB-Dist policy, where the CPUCap of the interfering workloads is decreased much more gradually than EB-Dist. HB-Dist allocates a higher CPU Cap to Hadoop (71) and lower CPU Cap to Linpack (50) since these are based on the I/O Ratio between the VMEs. For both these policies the PM always responds to a SLA violation messages within 5ms, therefore *DRX always detects and acts upon congestion in a timely manner.*

Essentially, EB and HB policies serve two respective methods for SLA satisfaction – (1) *fast-acting while not performing graceful degradation of workloads,* (2) *slow-acting while providing graceful degradation to other workloads.* This result highlights an important aspect of DRX: *multiple policies can be constructed and configured to meet the SLA values for applications.*

6.8.5 DRX Sensitivity to Resources

In order to evaluate the effect of resource usage patterns on the effectiveness of DRX, we use two more workload setups. In one setup we use an instance of Nectere along

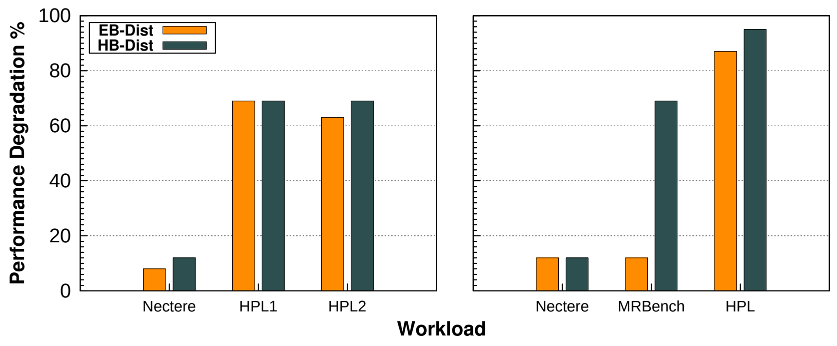


Figure 36: Comparison of DRX Policy Sensitivity with two different workloads sizes.

with 2 Linpack instances. In the second, we use an instance of Nectere along with the Hadoop MRBench application and a smaller data size for Linpack. We observe from Figure 36 that when the interfering workloads perform similar amounts of I/O both EB and HB policies behave equally well, however, since EB treats both VMEs similarly at all times, and applies the same cap simultaneously, it achieves a lower latency for Nectere. In the adjacent graph, HB becomes more aggressive than EB and it caps both Linpack and MRBench much more. This is because as HB performs the capping, it leads to oscillations in the VME that dominates the I/O Ratio ($> 95\%$). This forces HB to perform large amounts of cap alternately on the VMEs. This is not evident in Figure 32 as the difference in the I/O Ratio between Hadoop and Linpack is smaller. Therefore, we find that HB *is more sensitive to large swings in the I/O Ratio while EB is less sensitive to differences in the generated I/O.* Future policies will be extended with mechanisms to detect such oscillations, and to further limit their aggressiveness under such circumstances.

6.8.6 Limitations and Overhead of DRX

We show in Figure 37 that for two different workload layout (Asymmetric-1 and Asymmetric-2 from Figure 31), the DRX policies affect them differently. When there is a lot of interference in the Asymm1 layout, none of the policies can satisfy the

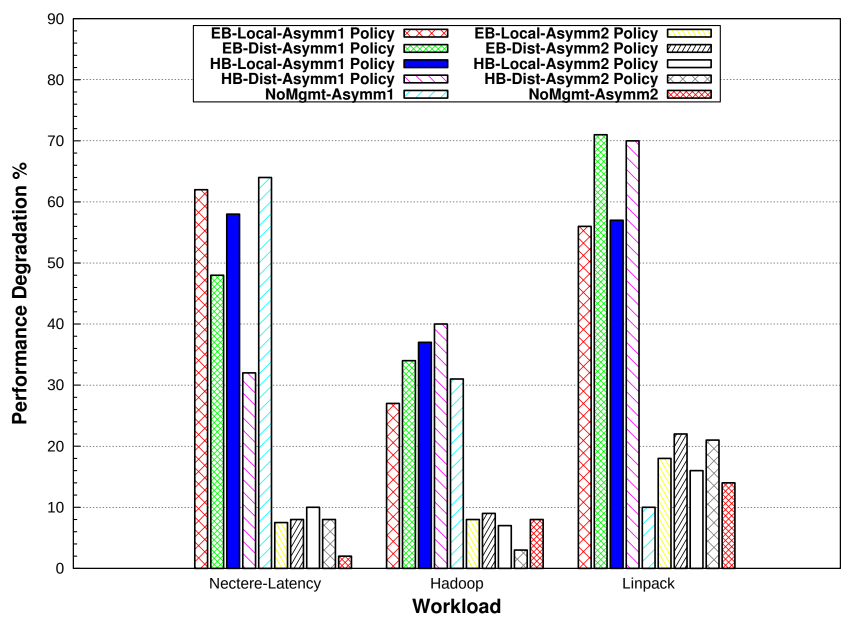


Figure 37: Performance of Policies (Local and Distributed) with Asymmetric Workload Layout. Asymm1 and Asymm2 refer to the types of workload deployment.

Nectere SLA. This is because, despite CPU capping, the VMs still generate sufficient I/O to cause congestion for Nectere. In this case, having more support from the hardware to control I/O would be very useful. In the Asymm2 case where Nectere has minimal interference, DRX ensures that other workloads are perturbed much less or not at all. Here, both Linpack and Hadoop perform very close to their baseline values. These results highlight *the limited utility of CPU Capping in extreme interference and also the low overhead caused by DRX components and their management actions.*

6.8.7 Effectiveness of Hardware Features with DRX

We perform initial experiments which leverage the Congestion Control and Quality of Service features built into InfiniBand hardware and how they can be used to improve latencies for the Nectere application. Figure 38 and 39 show the impact for two workload layouts, Symmetric and Asymmetric-1 and compare the performance between the best CPU Capping policies and Hardware Features Policies.

Both these figures highlight that the Static Enforcing Policy (StMax, StMaxQ)

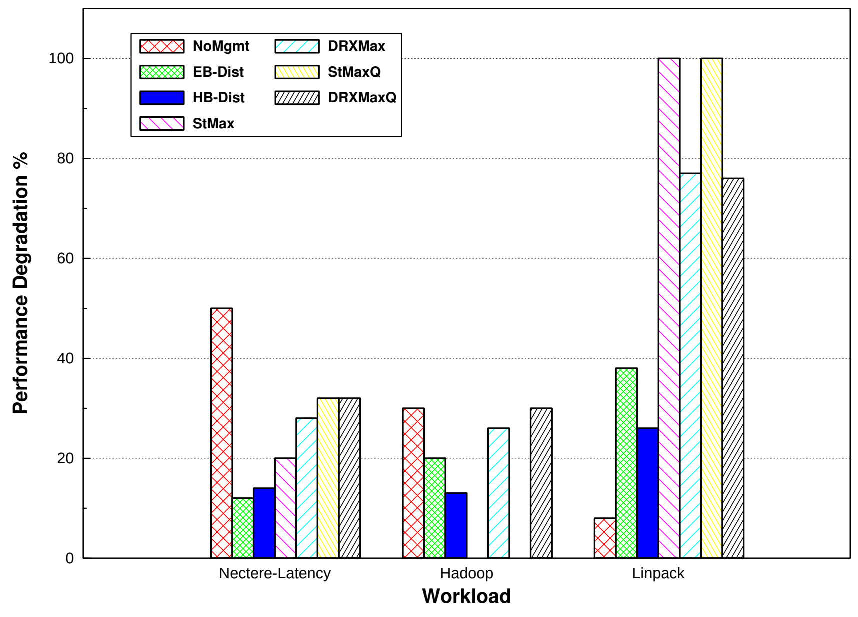


Figure 38: Performance Comparison of EB, HB and Hardware Policies for Symmetric Workload Layout.

outperform the DRX Enforced Policies (DRXMax, DRXMaxQ) as they limit the interfering workloads as soon as they start running and therefore have maximum impact on their performance. However, they cannot improve the Nectere latency below the 20% performance degradation which the CPU Capping policies easily achieve. This highlights one limitation of the hardware features that the enforcement granularity is not per VM but rather per SL which could be shared by multiple QPs across multiple VMs. Note, that in case of Hadoop and Static Policies, there is no corresponding column shown in the graph since Hadoop failed to start for these policies. One of the utilities of the hardware features is highlighted in Figure 39 where Congestion Control policies outperform CPU Capping policies and Nectere latency can reach 20% degradation. In such a workload layout where there is much more I/O than CPU Capping can limit, we can use the hardware features instead.

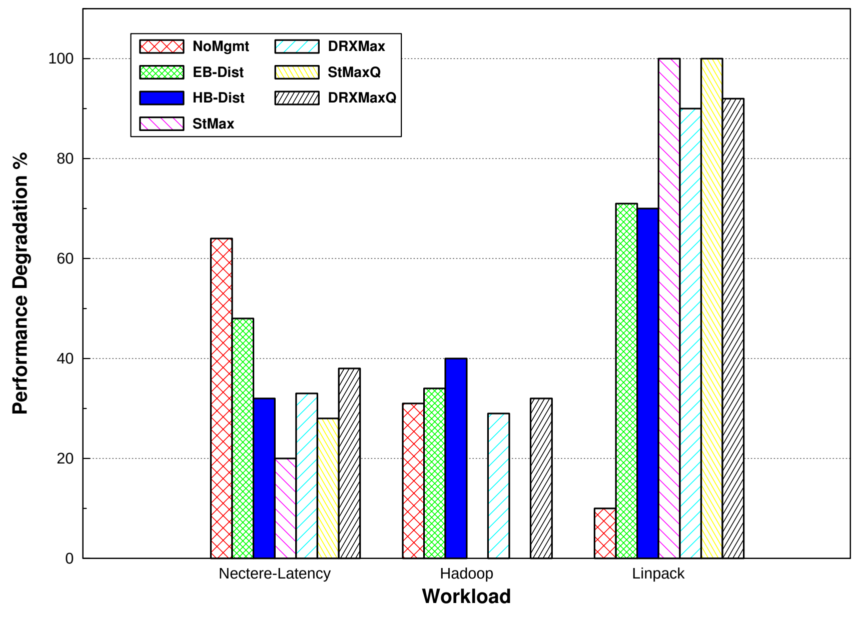


Figure 39: Performance Comparison of EB, HB and Hardware Policies for Asymmetric-1 Workload Layout.

6.8.8 Key Observations on Policy Effectiveness

In general, 'EB-Dist' and 'HB-Dist' provide that Nectere can meets its SLA in the Symmetric layout. In the Asymm1 layout, 'StMax' provides the best performance for Nectere, while in Asymm2 layout any of distributed or local pricing-based policies are useful in maintaining Nectere's SLA.

6.9 Chapter Summary

In this chapter we describe our approach called Distributed Resource Exchange or DRX which offers hypervisor-level methods to mitigate inter-application and distributed interference in SR-IOV-based cluster systems. We monitor VM Ensembles – a set of VMs part of a distributed application – which enables controls that apportion interconnect bandwidth across different VMEs by implementing diverse cluster-wide policies. We implement two classes of policies: one that assigns 'Equal Blame' to interfering VMEs and another that looks at how those VMEs are affected – 'Hurt-Based' policies, that show the feasibility of such distributed controls. Each class in turn contains policies that apply the Price changes in a distributed or local manner, and that only use the software- or hardware-based platform controls. With DRX, we show that latency-sensitive codes can be protected against sets of distributed applications, when they experience 'hurt' only when hurt is present.

The results demonstrate that DRX is able to maintain SLA for low-latency codes to within 15% of the baseline by controlling collocated data-analytic and parallel workloads. Limitations of the DRX approach are primarily due to its current method to mitigate interference, which is to 'cap' the VMs that over-use the interconnect and cause 'hurt'. We also show initial results with leveraging hardware-based congestion and quality of service controls for mitigating congestion for latency-sensitive workloads.

CHAPTER VII

ENFORCING LATENCY PROPORTIONALITY IN SR-IOV INFINIBAND CLUSTERS USING OPENFLOW CONTROLLERS

Previous chapters have highlighted the necessity to provide isolation, fairness and performance guarantees to collocated sets of latency-sensitive and communication-intensive applications. While, the mechanisms used are general in nature, the management platform cannot be integrated into existing network management platforms to be generally applicable.

In this chapter, we first highlight the motivation to add such mechanisms for current network management platforms and how they can utilized to provide a new fairness property called 'Latency Proportionality' for collocated latency-sensitive applications. We use our Virtual Currency Abstraction mechanics to build a dynamic feedback-based controller, Sphinx, that performs network allocations through hardware-based controls. Sphinx is built on top of the OpenFlow-based Floodlight controller and administers a SR-IOV InfiniBand cluster to provide latency guarantees.

7.1 Importance of Latency Proportionality

Performance-critical flows are becoming increasingly common in datacenters where networks with new features like Remote Direct Memory Access (RDMA) [52, 54, 1] and Single Route I/O Virtualization (SR-IOV) [2, 32] are being deployed. These flows are part of new application classes like financial [126, 123], VoIP [74], Memcached [59, 61], Hadoop [55] and Video Streaming [17, 35] which have specific performance constraints that rely on the low latency and high bandwidth capabilities afforded by these features.

Another trend common to datacenters is the deployment of Software-Defined Networking or SDN [25, 69, 82] solutions to simplify network management. Current cloud computing platforms can achieve network-wide visibility through a centralized logical controller [37, 97, 20]. The various flows between virtual machines (VMs) or applications can be managed through the centralized interface and network access policies can be constructed accordingly. This logical separation of policy definition from enforcement allows for specification of policies for performance-critical flows. However, there are key challenges related to identifying flow requirements, enforcing flow isolation as well as monitoring flow performance that have to be addressed in order to improve server consolidation of performance-critical flows.

Currently, specification of flow requirements [10, 75] is more commonly treated as a division of network throughput between collocated flows based on some fairness conditions [104]. While this works well for throughput-intensive workloads, this still does not apply in the same manner to the latency-sensitive workloads which cannot meet their guarantees. As evident from recent work [40, 7, 63], latency-sensitive workloads need more co-operation from the network/host to reduce the 'delay' with which its packets get sent on the wire. By providing acceptable bounds to this delay, current network management solutions can better service performance-critical applications.

Second, with RDMA-based fabrics deployed in datacenters, there is an increasing need to manage these type of flows collocated with other TCP-based flows. Due to this heterogeneity, network resources must be balanced in order to bound the delay for the performance-critical flows.

Third, since network and resource management solutions cannot apriori know the impact of collocated workloads, they must perform continuous monitoring of these flows to check for unbounded delays. Thus, as a result of missing the delay guarantees they must orchestrate network resources to let the flow meet its delay bound.

Recent work [98, 63, 113] has addressed some challenges of scheduling individual low latency flows on shared networking infrastructure. Other efforts show the advantages of coordinating flow schedules across the network through various virtual abstractions [4, 15, 42] as well as minimizing flow interference between tenants [14, 56, 40]. Also, efforts to find interference effects and resource requirements can be obtained by sandboxing applications [31, 99, 80], however, such efforts only deal with CPU and Memory constraints while ignoring the network component. Thus, there are still challenges related to dependence of intra-application flows as well as heterogeneous types of flows that need to be addressed to schedule performance-critical workloads correctly.

In this chapter, we address the problems discussed above by creating a management solution for high performance networks called Sphinx, which manages the performance interference of collocated performance-critical and data-intensive flows. We identify the permissible level of interference through a Proportional Fairness-based [81] property called 'Latency Proportionality' which identifies the correct fairness and guarantee for such flows based on user-defined policies. Sphinx enforces this property through our 'Virtual Currency Abstraction' described in Chapter 5 as well as orchestrating the network via various host-specific and cluster-wide network resource controls. We use an existing network controller to build these abstractions and highlight the need for such management actions through a combination of user-defined policies and latency-sensitive workloads.

Broadly, this chapter makes the following key contributions:

(i) Definition of a new network property, *Latency Proportionality*, that specifies the performance bound on the collocated application flows on the network based on their latency-sensitivity.

(ii) Dynamic network resource allocations using our *Virtual Currency Abstraction* and network hardware features.

(iii) An implementation, *Sphinx*, of the above mechanisms, that can enforce flexible policies through the Floodlight OpenFlow Controller.

(iv) Evaluation of Sphinx on a SR-IOV-based InfiniBand cluster with a mix of latency-sensitive and data-intensive workloads. Sphinx utilizes the hardware features of InfiniBand fabrics to allocate network resources for flows.

The rest of this chapter is organized as follows. Sections 7.2 and 7.3 describe the key motivations for developing Sphinx and the technologies used by Sphinx respectively. In Section 7.4 we define our Latency Proportionality network property. We describe our Pricing Resource Control Model in Section 7.5. Section 7.6 describes the implementation of Sphinx. We provide an evaluation of Sphinx in Section 7.7, and we summarize in Section 7.8.

7.2 Challenges in collocating diverse workloads

In this section we show that with SR-IOV fabrics there are challenges when collocating workloads with mixed SLA requirements and using mechanics provided by the fabric mitigate the impact, are insufficient by themselves. We begin by showing initial results of running both throughput and latency-sensitive applications on our testbed (detailed in Section 7.7), comprising of 12-core nodes connected by a 40Gbps Single Route I/O Virtualization (SR-IOV) enabled InfiniBand network.

7.2.1 Impact of Workload Characteristics

For a RDMA-based application the network request bypasses the OS and therefore, the request size determine the workload characteristics application [110, 58]. Thus, by providing larger request sizes the application can grab a higher share of the bandwidth.

We demonstrate this behavior in our testbed using two instances of Memcached [59] to request data of sizes 16K and 512K respectively. Figure 40 shows that there is an

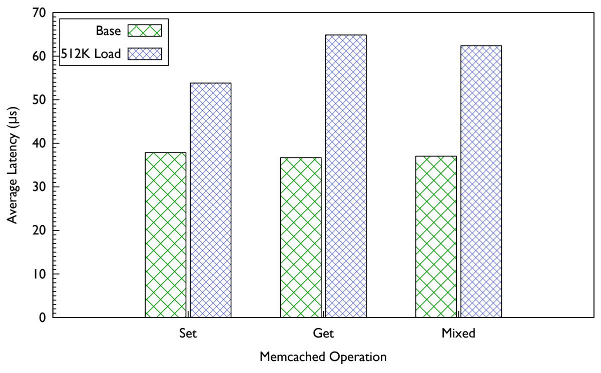


Figure 40: RDMA-based Memcached running with another Memcached instance using larger message sizes.

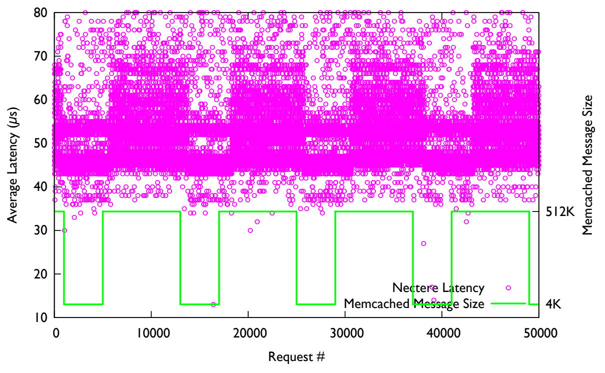


Figure 41: Nectere running with a Memcached instance which alternates between small and large message sizes (shown by green line). Nectere's latency also increases or decreases when the Memcached message size changes.

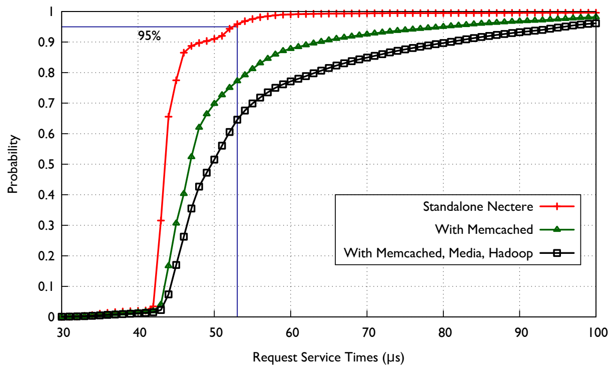


Figure 42: CDF of a Latency-Sensitive application with other Cloud workloads.

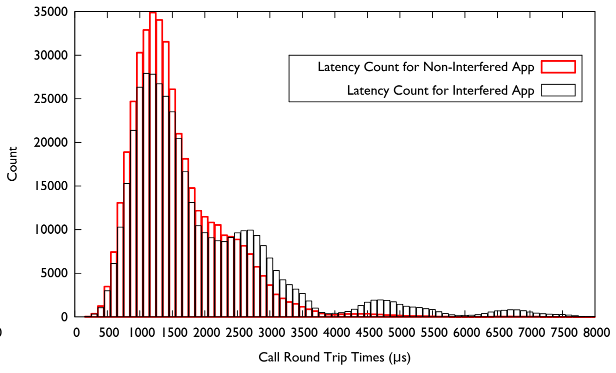


Figure 43: Latency Distribution of a VoIP application with Hadoop TeraSort workload.

impact on the response time for the 16K client even though the requests are being serviced by two different but collocated server VMs. We modified a RDMA-Memcached client to perform 'Get' operations of different message sizes (4K and 512K) for 1K iterations alternatively. Figure 41 shows the impact on a collocated Nectere [45] application's latency, which tracks the rise and fall in Memcached message sizes. The latency increases by almost 40% when the Memcached message size is 512K.

7.2.2 Impact of Workload Collocation

We run a few cloud-based applications like Hadoop, Memcached, Media-Streaming [35] with other Latency-constraint applications like Nectere and Voice Over IP in our testbed. We show the impact of collocating these cloud-based applications with Latency-sensitive applications in Figures 42 and 43.

Figure 42 shows the CDF of the Nectere Request Times when running (i) standalone, (ii) with Memcached, and (iii) with Memcached, Hadoop, Media Streaming Benchmark. Figure 43 shows the Latency distribution for a VoIP/SiPP benchmark when running standalone versus collocated with Hadoop. Both these figures demonstrate that even on modern interconnect networks which support RDMA and OS-Bypass features, server consolidation can lead to reduced performance for workloads. This is mainly due to the sharing of the network link between the VMs on the host and as a result some VMs get a lower share of the bandwidth than their SLA requirements. Since the network card that schedules packets from each VM on the link is not aware of the VM SLA it does the scheduling naively, i.e. to improve the bandwidth utilization.

7.2.3 Configuring Network for SLA

Since for RDMA-based fabrics the NIC is responsible for servicing application requests directly, the firmware on the NIC must support management of applications. This management could include bandwidth rate limiting or stopping/starting flows in the network. This would help in alleviating some of the workload characteristics impact on other applications.

With respect to InfiniBand networks there are two capabilities - Quality of Service (QoS [6]) and Congestion Control (CC [117]) - which can be used to reduce interference between applications. Briefly, QoS allows us to prioritize communication from one application over another by assigning weights to each application. CC allows us

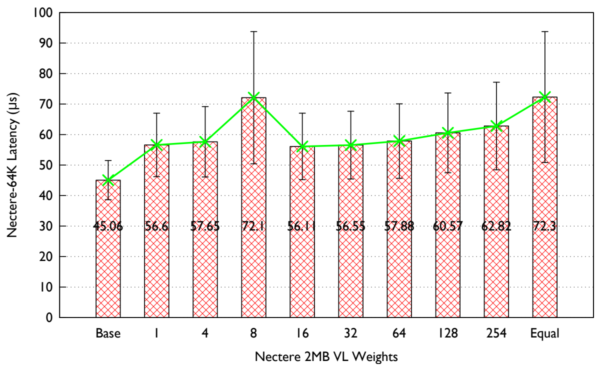


Figure 44: Using InfiniBand QoS to control Nectere workload. The values on the X-axis denote the QoS weight assigned to the 2MB Nectere instance. The 64KB Nectere instance is assigned a higher priority service level than the 2MB one. In the Equal case both instances are at the same high priority service level and equal weights.

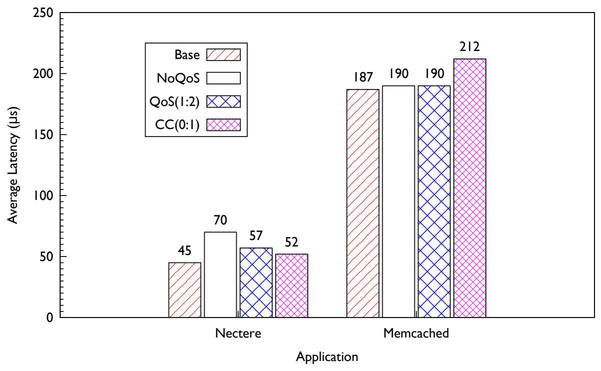


Figure 45: Using both InfiniBand QoS and CC mechanisms to control the interference from the Memcached workload. Values on the bar denote the latency of the respective application. The QoS values denote the weights assigned to each application. The CC values indicate the units of delay added to the queue processing of each application.

to add delays in the servicing of requests sent from applications. We describe these in more detail in Section 7.3. The NIC is responsible for enforcing these capabilities when it is configured using the Subnet Manager (SM).

We explore these options in trying to reduce interference in applications used above in this section. In Figure 44 we use QoS to reduce the interference between two Nectere instances. While the QoS can reduce the interference to a certain degree, it is harder to predict what the reduction in the value would be as well as how it will affect the variation. For example, with a weight of 8 for the 2MB Nectere instance, the interference effect is the same as the one for Equal weight.

In Figure 45 we show the effects of using QoS and CC on two co-located applications with different workload characteristics to reduce interference. With either QoS and CC there is an reduction on the latency value but with CC the impact on the Memcached instance is more. Therefore, there must exist some combination of QoS and CC that might reduce interference but the impact is not as much on other applications.

7.2.4 Observations

The above experiments demonstrate that while collocating disparate workloads is good for increasing link and server utilization there are still some challenges in how the link bandwidth should be divided in order to satisfy applications' SLAs. One, in order to meet VM/application SLA the network should be configured correctly and in the case of hardware virtualized and RDMA-based NICs, this entails finding the right values for the hardware-based rate-limiting mechanisms. Therefore, there must be some *mechanism to convert the application SLA values to a network-specific rate-limit value.* Two, due to the work-conserving property for existing network management, both latency-sensitive and bandwidth-based flows are treated equally. This actually leads to more delays in processing for latency-sensitive flows. Thus, to reduce this delay, the network should be configurable such that *latency-sensitive flows are given a higher preference before other flows.* Three, as workload characteristics change the impact on collocated workload is affected which may increase the amount of congestion caused. Thus, as a result the corresponding hardware rate-limit value may be incorrect in satisfying the application SLA. This implies that we must *monitor applications' current performance and update the rate-limit values continuously.* Four, when we adding delays or priorities to a single VM belonging to a cluster-wide distributed application, it impacts the entire application SLA as well. Therefore, it is crucial to find the *right network configuration values for all the nodes hosting the application components.* Five, the tuning of the network configuration components is crucial as well to find the right combination such that applications can meet their SLA while *other applications are not extremely penalized* which might lead to very under-utilized networks.

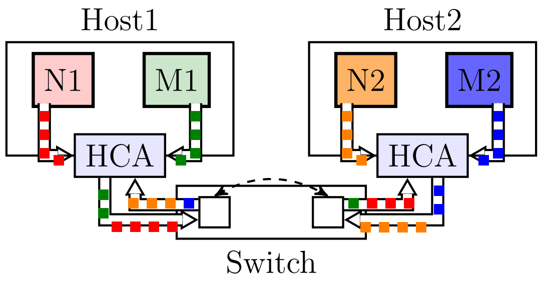


Figure 46: N1, N2, M1, M2 represent VMs or application components for two applications, Nectere and Memcached respectively. The colored packets represent the corresponding source flow. This represents the key idea behind Latency Proportionality, that is to prioritize packets from particular flows on the link.

7.3 OpenFlow-based Network Controllers

Sphinx relies on several technologies that motivate its design. We describe the hardware virtualization techniques in Section 2.5 as well as the hardware network mechanisms in InfiniBand in Section 6.5.2. Here we describe additional technologies relevant only to Sphinx.

Software-Defined Networking provides a separation of the control and data plane of the networking infrastructure into configurable software components. The OpenFlow protocol [82, 37] provides a specification for how these separate components can interact. The OpenFlow controller is the software component that controls the network hardware to perform datapath operations based on network policies entered by the operators. This simplifies network management by presenting a uniform OpenFlow-based interface even if the hardware itself is heterogeneous. In a virtualized environments, the controller manages both virtual and physical switches and collects statistics from the switches based on the OpenFlow protocol.

7.4 Latency Proportionality

7.4.1 Definition

We define 'Latency' as well as Service Level Objective for an application as the time measured between consecutive requests sent. To achieve the required latency the application must receive a set of I/O resources corresponding to flow credits, scheduling slots, switch buffers on the network link. When applications are collocated these flow credits and slots are shared resulting in latency increases and thus performance reduction. Due to differing I/O characteristics these network resources end up being shared in an 'unfair' manner. This sharing is worse for OS-bypass and RDMA networks as there is little control at the OS-level apart from CPU Capping [113].

The cause of this interference is well-studied and arises from some packets [112, 40] of a more latency-sensitive flow being delayed by packets from other flows. In order to reduce the delay packets from lower latency (or more performance-critical) flows should be scheduled on the network link first. We illustrate this in Figure 46, where flows from two latency-critical applications must be prioritized on the link for all components of the applications.

However, the amount of delay to be reduced is not known since achieving a delay of zero for a flow would imply degrading the rest of the flows by almost 100%. We need a metric to provide a 'fair' and 'expected' performance for collocated applications, such that this delay between packets of each application is bounded. The delay can be summed as 'Performance Loss' for the flow SLO for a request that is sent. Thus, the 'Fairness criteria' can be defined in terms of expected Performance Loss to the flows which can be similar to the notion of 'Proportional Fairness' [81] well-established before. We call this criteria *'Latency Proportionality' or 'LP'* and it allows us to define relative performance degradation between collocated flows. For the four flows shown in Figure 46 we can define LP for the pairs of collocated flows as:

$$LP(N1, M1) = \frac{PL_{N1}}{PL_{M1}} \quad LP(N2, M2) = \frac{PL_{N2}}{PL_{M2}}$$

In this rest of this section, we show how we can achieve LP for such collocated flows deployed on a RDMA-based cluster using hardware mechanisms and outline our observations from these experiments.

Table 8: Comparison of Latencies (in μs) achieved by Nectere and Memcached workloads when running in four configurations. All cell values are in Nectere : Memcached format. PL refers to the Performance Loss experienced by the application and is shown as a percentage change from the Base values for the BP and LP configurations.

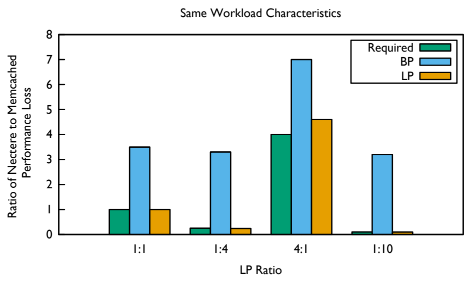
LP Value	Base	Collocated	Collocated + BP	Collocated + LP	**% PL Ratios**	
					BP	**LP**
1 : 1	48 : 187	64.1 : 196.7	63.7 : 204.5	59.2 : 230	33% : 9.4%	23.3% : 23%
1 : 4	48 : 187	64.1 : 196.7	63.3 : 205.1	54 : 284	32% : 9.7%	12.5% : 52%
4 : 1	48 : 187	64.1 : 196.7	65.8 : 196.9	63.6 : 200.3	37% : 5.3%	32.5% : 7%
1 : 10	48 : 187	64.1 : 196.7	63 : 205.5	52.4 : 364	31% : 10%	9% : 95%

Table 9: Comparison of Latencies (in μs) achieved by different Nectere and Memcached workload characteristics with the same configurations as above.

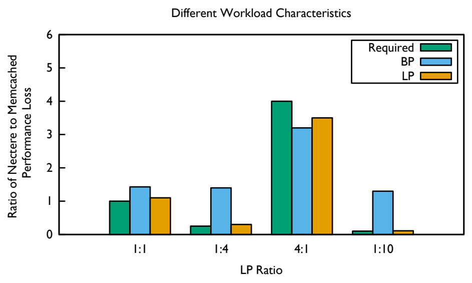
LP Value	Base	Collocated	Collocated + BP	Collocated + LP	**% PL Ratios**	
					BP	**LP**
1 : 1	100.5 : 99.5	121.3 : 112.8	116.3 : 110.4	116 : 113.5	15.7% : 11%	15.4% : 14%
1 : 4	212 : 187	238.2 : 191.4	225.6 : 195.6	222.3 : 215.9	6.4% : 4.6%	4.9% : 15.9%
4 : 1	48 : 46.7	57.4 : 51.1	60.7 : 50.6	60.8 : 50.3	26.5% : 8.3%	26.7% : 7.7%
1 : 10	212 : 99.5	223.4 : 112.6	223.5 : 103.8	219.5 : 132.8	5.4% : 4.3%	3.5% : 33.5%

7.4.2 Enforcing LP via hardware controls

We use the technologies and applications described in Section 7.3 related to InfiniBand to control the performance losses experienced by collocated applications. The two applications we use are Nectere [45] and RDMA-based Memcached [59]. We configure each application in terms of request size, computation performed in order to show various cases for Latency Proportionality values. We run each application component (server, client) in separate VMs on different hosts on our testbed (described



(a)



(b)

Figure 47: Comparison of BP-based and LP-based performance loss ratios for collocated Nectere and Memcached workloads. The configurations are for Table 8 and 9 respectively.

in Section 7.7) for each LP configuration. For LP value we run the applications - one, without any resource controls to show the collocated performance, two with resource controls to show we can meet the required LP values. We use the hardware controls - Quality of Service and Congestion Control present on InfiniBand HCAs to control the set of resources allocated to each application. Here, each application is allocated a different Service Level (SL) for enabling the hardware controls.

Tables 8 and 9 provides the latency values for Nectere and Memcached when they are running in four configurations: (i) Base (or standalone), (ii) Collocated, (iii) Collocated + BP, where we configure the IB QoS weights in the inverse ratio of LP specified, (iv) Collocated + LP, where we configure both IB QoS and CC values in order to meet the required LP value. Figure 47 shows the LP value achieved for four different LP configurations from the Tables 8, 9 and how we can achieve them using the right settings for the hardware controls.

7.4.3 Observations

From the above graph and table, we can summarize the results as follows. One, when partitioning the network from a complete bandwidth perspective it is hard to achieve LP as seen from Figure 47. This is due to differing I/O characteristics of the collocated flows as well as how work-conserving the hardware platform is. When the

I/O characteristics are equal as in LP 1:1 (100 us), LP behaves very similar to BP. Two, each hardware control setting corresponds to a set of resources provided to the application. These sets of resources does not directly correspond to the LP, for two reasons mentioned above. Three, the controls established here are static controls, while more generally we need a dynamic approach that takes into consideration the current performance loss experienced by the application and makes the corresponding change in hardware controls to achieve the Latency Fairness Property. The next section describes our control model to dynamically adjust the resource controls based on the performance loss experienced by the flows.

7.5 Dynamic Hardware controls via Congestion Pricing

When application flows interfere with each other, the network resources allocated to each flow may need to be changed based on the performance loss experienced by the flows. Since we do not have apriori knowledge of workloads and the impact of resource controls, we use our 'Virtual Currency Abstraction' to abstract the relationship between the performance losses experienced and exact resource controls. This also allows us to deal with the dynamic nature of an application's performance through 'Price Changes' on collocated applications that result in resource allocation changes. We combine both these key ideas into our 'Pricing Control Model' that can dynamically orchestrate network resources to meet a particular performance loss or LP value. We next discuss several key aspects of this model.

7.5.1 I/O Control Model

Even though we do not rely on apriori knowledge of workloads, we need to know the impact of changing resource controls on application performance. This dictates the right amount of control to set in order to increase or decrease application performance loss. One of the approaches to apply control theory to system modeling is to use a black-box approach [102], where we vary the resource controls and see the

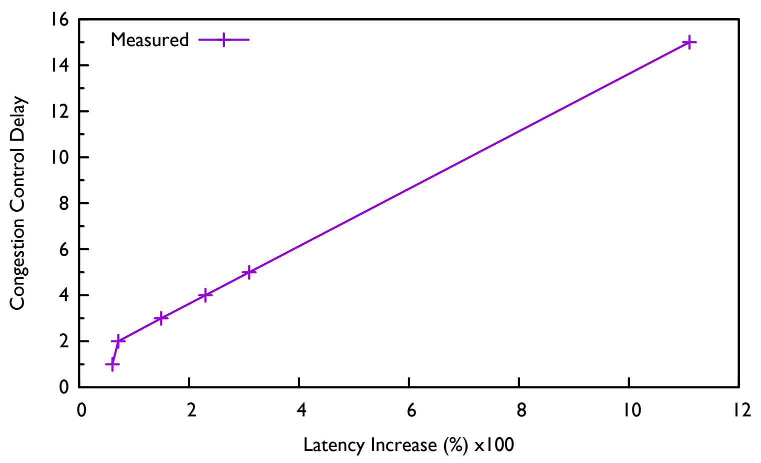


Figure 48: Impact of Congestion Control delays on RDMA Send Average Latency for a 512KB buffer.

corresponding impact on the application performance. Instead of creating a model for each RDMA-based application, we measure this impact on the individual RDMA operations which are the basic building blocks for such applications. This allows us to generalize the impact of the resource controls while at the same time reduce the possible set of combinations to model. 'RDMA Send' is one of most common operations used and we study the impact of using InfiniBand Congestion Control delays on its latency.

We utilize a RDMA Send microbenchmark that measures the latency of the operation for different buffer sizes. This benchmark is run between two VMs on separate hosts (we describe our testbed in Section 7.7) and we keep increasing the delay to measure the latency impact. The delay is changed by interacting with the InfiniBand Subnet Manager (SM) through Management Datagrams (MADs) to specify the host HCA and the delay value required. The delay value is converted to an IRD value by the HCA for a QP as we describe before in Section 6.5.

Figure 48 shows the impact of the congestion control delays on a RDMA Send microbenchmark using a 512 KB buffer size. As we see from the graph, this implies a mainly monotonically increasing relationship between the latency increase and the

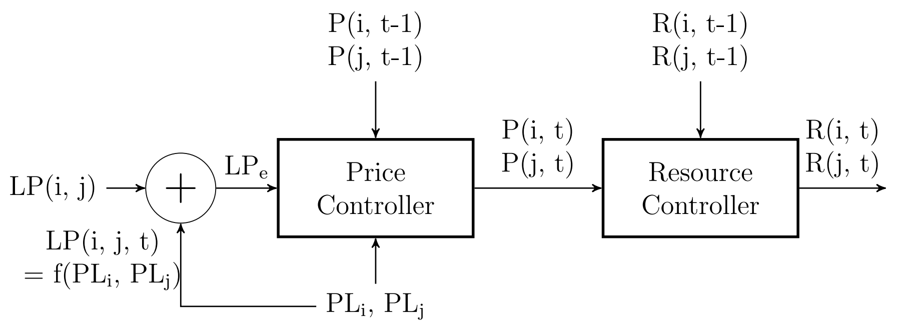


Figure 49: Sphinx Two-Stage Network Resource Controller.

delay value. Therefore, we can easily use a linear regression approach to find the exact relationship between the latency increase or performance loss with changing delay. The following equation captures this and we can use the coefficient of the equation to find the right delay value given the performance loss desired. This linear relationship exists for other buffer sizes as well though there was no effect on latencies of buffer sizes of 4KB and smaller.

$$\Delta CCDelay = 1.28 * \Delta Latency \tag{8}$$

This implies that a linear control model will allow us to find the right congestion control delay given the latency percentage increase desired. However, as the number of delay values are limited and small values can have a large impact on latency (for example, when delay = 15), we still need to flexibly and dynamically allocate resources, which we describe next.

7.5.2 Pricing Control Model

We use our 'Virtual Currency Abstraction' to provide a dynamic and flexible resource abstraction to manage network resources. We perform our price changes in a dynamic fashion based on the monitored values from the applications such that it can meet the desired latency objectives, i.e. the Latency Proportionality property we described in Section 7.4. Essentially, our 'Pricing' mechanism finds the right response-time curve for a flow's latency to the change in network resources.

There are two goals that the Currency mechanism provides: one, a dynamic method to control over time the LP property desired and two, a single control point for allocating resources for different applications, i.e. abstract the actual physical resource control used.

These requirements allow us to construct a controller that takes as its input as the error between the LP values required and the desired value for the collocated set of flows. The final output of the controller will be the right resource settings that allow us to reach the LP value. However, since we do not know apriori the right resource controls to use we must perform this in two stages. One, we find the 'Price Change' based on this error which can be negative or positive. Two, this 'Price change' then dictates the right resource control to use, i.e. IB QoS, Congestion Control, CPU Capping [113] or Linux *qdisc* mechanisms.

We show one such controller in Figure 49 where we compute the New Price based on the current Price for the flow, the error in the desired LP value as well as the current performance experienced by the collocated flows. We can define the control equation for a flow i using our Price Controller stage as follows:

$$P(i,t) = f(LP_e, P(i,t-1), PL_i)$$

Similarly, our Resource Controller stage accepts the updated Price changes and previous resource settings to output the settings to be applied to the network as follows:

$$R(i,t) = f((P(i,t) - P(i,t-1)), R(i,t-1))$$

We model our Price Controller on an integral controller where we keep track of all the errors measured for the target LP value. This allows us to make sure our Price changes are an effect over time rather than at each time quantum. This also smoothens out the impact of the price changes to help reach the right LP value.

Furthermore, we expand our control equation as follows:

$$P(i,t) = K_I * \int_0^t LP_e(t)dt \tag{9}$$

where, K_I is our integral gain.

We model our Resource Controller on the linear property as a Proportional Controller as defined by Equation 8 as follows:

$$\Delta R(i,t) = K_P * \Delta P(i,t) \tag{10}$$

where, K_P is our proportional gain.

Through both these equations 9 and 10 we can meet our goals for the Price Control model which we defined above. In the next section we describe how we use this Pricing model to enable the LP property within a software-defined network controller.

7.6 Sphinx OpenFlow Controller for Enabling LP

In this section we describe Sphinx, our OpenFlow Floodlight-based [37] network controller to administrate a SR-IOV InfiniBand cluster using the Pricing Control Model described earlier to help collocated flows maintain Latency Proportionality.

7.6.1 Overview

Figure 50 shows the overall architecture of our Sphinx OpenFlow Controller to orchestrate a SR-IOV InfiniBand-based virtual cluster. We modify an existing networking controller, *Project Floodlight* [37], to implement the mechanisms and methods described before in this chapter.

Overall, the main goal of Sphinx is to achieve the desired Latency Proportionality value by continuously monitoring the latencies of the workloads and making the appropriate changes using the Pricing and Resource Controllers to change network resources. The main components Pricing and Resource Controllers interpret the monitoring data sent by the IBMon tool deployed on each host. These controllers are

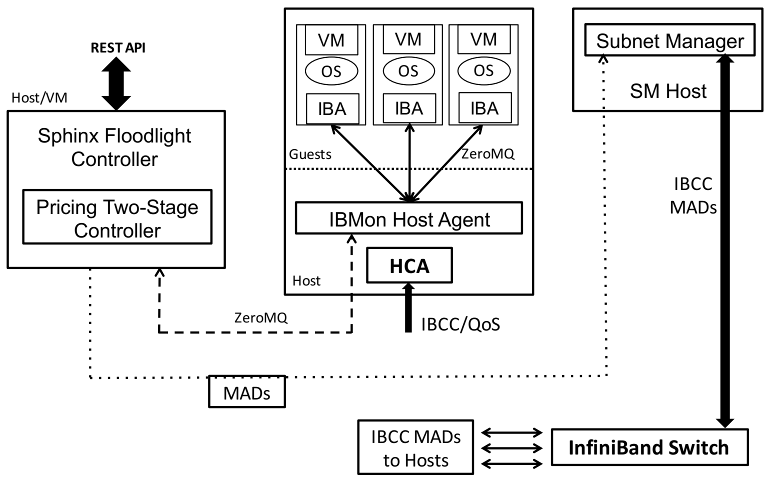


Figure 50: Enforcing LP with OpenFlow Controller in SR-IOV InfiniBand Cluster.

deployed as a module within Floodlight. We use ZeroMQ [143] as the out-of-band communication method between IBMon instances and controllers. Floodlight also exposes REST APIs to allow users to set flow properties and latency proportionality values. To effectively orchestrate and manage the network, we divide the Sphinx workflow into various components described next.

7.6.2 Flow Specification

Applications are divided into flows such that each flow gets its separate SLO. This can be done apriori based on the requirements of the user. Each flow is identified with its flow SLO, the type of flow (RDMA or TCP) and the different endpoints of the flow. A model described here [75] can be used to mark specific communications of an application as latency-sensitive as required and the remainder can be tagged as throughput-based. For throughput-intensive applications we allow the user to specify a minimum bandwidth guarantee which we convert to a network latency value for sending a MTU sized packet. This allows us to treat throughout-based traffic as a LP flow.

7.6.3 Specifying LP through Policies

Rather than allow users to specify the exact LP value we describe certain general policies that pertain to behaviors that users might want flows to achieve. For example, when the user wants to establish a 1:1 LP, he/she can pick the 'Fair' policy. These policies can be specified using the same APIs we expose for setting the flow model. While these policies are built into Sphinx, the mapping of the LP ratio can be changed as required. This allows us to dynamically change the mapping if required to give the notion of differentiated performance. We use these policy definitions to define certain minimum guarantees for workloads using the bandwidth proportionality notion we define in Section 7.4.

7.6.4 Monitoring Heterogeneous Flows

We identify both RDMA (QP-based) flows as well as TCP/IP (IPoIB) flows using Sphinx. As applications continue to run on the cluster we monitor each flow using our IBMon tool. Each host in our cluster contains an IBMon instance that measures both IPoIB and RDMA flows to find the amount of data is sent (based on information sent by Sphinx). Each VM contains an extremely lightweight IB Agent that records the relevant memory address of QPs, CQs created by the VM and reports it to the main IBMon agent running in the host via ZeroMQ messaging methods. For RDMA-based flows IBMon uses Libvirt [76] memory-mapping functions to map this guest memory into its address space to track WQEs added to the QPs and completions added to the CQs. The WQE contains a 'byte count' attribute to indicate how much data was sent and the CQE contains the same value in its 'byte count' field when the request completes successfully. This allows us to track the amount of data a QP sent. For IPoIB flows, we use the port statistics (TX bytes and packets) that are present in the guest and return those values to our IBMon agent.

7.6.5 Orchestrating Network Resources to manage LP

Sphinx relies on the 2-stage controller we described in the previous section to help maintain LP. The advantage of the 2-stage controller is to allow us to substitute various different resource control knobs as required by the application monitored. Since we monitor heterogeneous set of flows, each type of flow would need a different control knob to change its allocation.

Sphinx is deployed for InfiniBand networks, therefore, we provide a Resource Controller that utilizes IB hardware mechanism - Quality of Service and Congestion Control to control RDMA-based flows. The performance of the controller depends on the integral gain and the proportional gain values used in the equations to provide a stable LP output. Based on manual experimentation using Nectere and Memcached workloads (using configurations from Table 8) and InfiniBand CC Delays, we found that using $K_I = 4$ and $K_P = 0.25$ as the gain values gave us the most stable values.

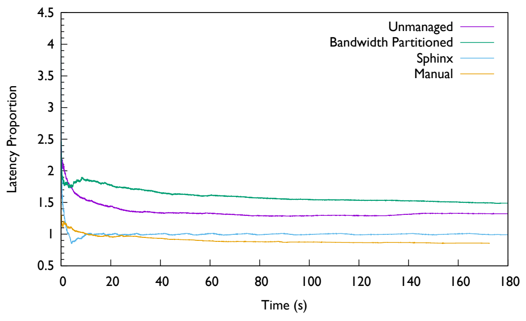
7.7 *Evaluation*

7.7.1 Testbed

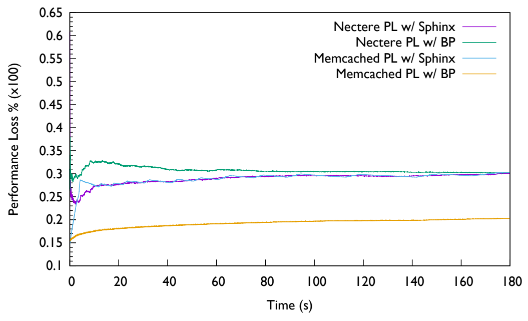
Our testbed consists of six Relion 1752 Servers. Each server consists of dual hexa-core Intel Westmere X5650 CPUs (HT enabled), 40Gbps Mellanox QDR (MT26428) ConnectX-2 InfiniBand HCA, 1 Gigabit Ethernet and 48GB of RAM. Our host OS is RHEL6.5 OS with KVM and the guest OS' are running Centos6.1. Both the host and guest are configured with recent Mellanox software to enable SR-IOV support (upto 16 VFs) for the HCAs. The IB cards are connected via a 36-port Mellanox IS5030 switch.

7.7.2 Workloads

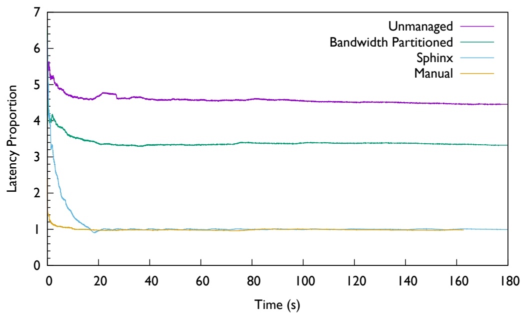
We use a combination of microbenchmarks, RDMA- and TCP-based benchmarks to evaluate the utility of Sphinx in such clusters. Nectere, is a financial application [45]



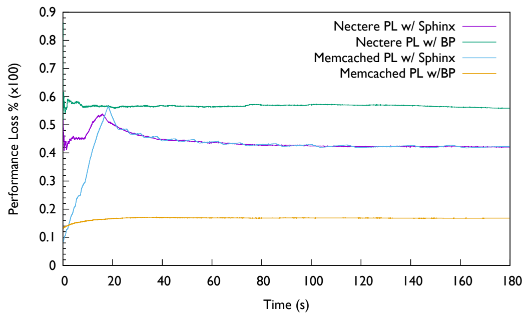
(a) Latency SLOs: 100.5us vs 99.5us



(b) App Performance Loss - With Sphinx and Bandwidth Partitioned



(c) Latency SLOs: 48us vs 187us



(d) App Performance Loss - With Sphinx and Bandwidth Partitioned

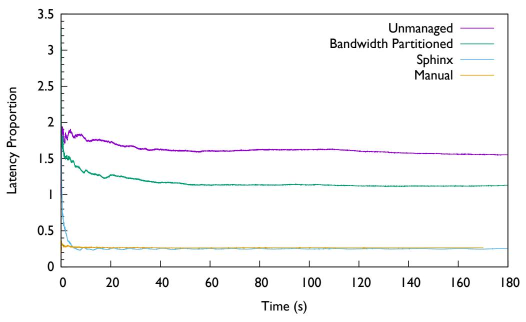
Figure 51: Achieving LP 1:1 for collocated Nectere and Memcached applications.

that was developed previously by us. RDMA Memcached [59] is a low-latency version of Memcached that leverages the Verbs API designed for high performance RDMA fabrics.

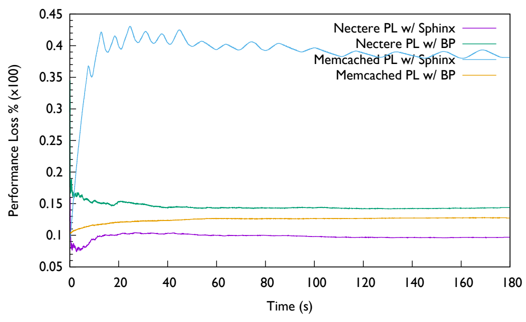
We compare LP values, Performance Loss and Application Latencies achieved with three other resource management policies - Unmanaged, Bandwidth Proportioned and Manual. Unmanaged means no resource management is being performed and both applications share the same IB QoS service level. We define the Bandwidth Proportioned (BP) and Manual policies in Section 7.4. The Manual policy implies the best division of resources that can be achieved to meet the LP.

7.7.3 Achieving LP through Sphinx

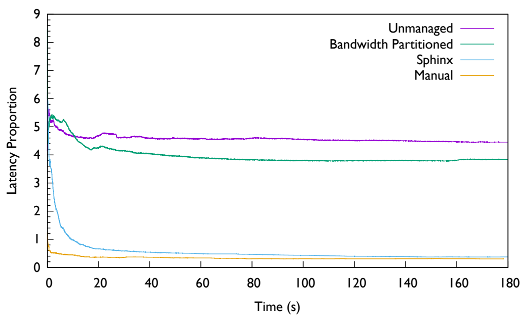
Figure 51 and 52 show three aspects when achieving LP for different latency SLO combinations for collocated Nectere and Memcached applications. One, Figures 51(a), 51(c),



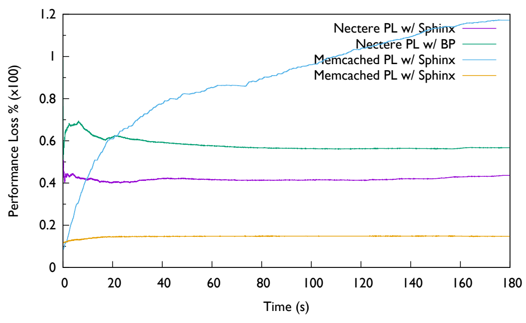
(a) Latency SLOs: 212us vs 187us



(b) App Performance Loss - With Sphinx and Bandwidth Partitioned



(c) Latency SLOs: 48us vs 187us

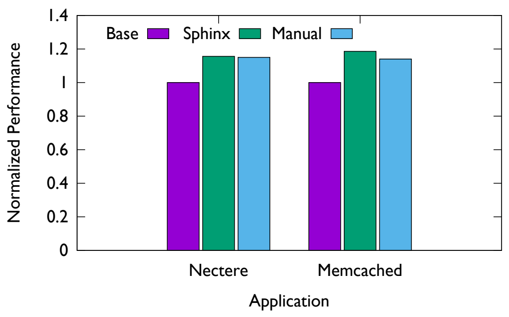


(d) App Performance Loss - With Sphinx and Bandwidth Partitioned

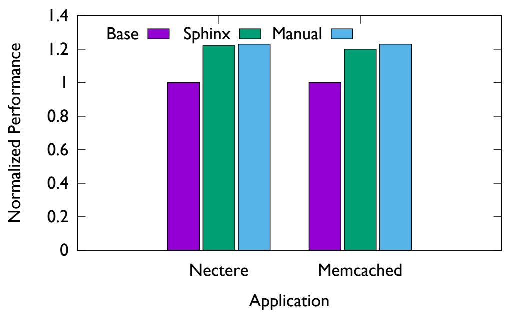
Figure 52: Achieving LP 1:4 for collocated Nectere and Memcached applications.

52(a) and 52(c) show how Sphinx changes the Latency Proportionality to required value for the collocated applications and two target LP values - 1:1 and 1:4. The LP values achieved by Sphinx are closest to the Manual values when compared to the Unmanaged and BP cases. This highlights the control model of Sphinx where it continuously updates the resource management to keep the LP to required LP.

Two, since LP tracks the ratio of the performance loss of the application, i.e. difference in latency experienced from the SLO, we also show the continuous performance loss experienced by the respective applications in Figures 51(b), 51(d), 52(b) and 52(d). In most of the cases, Sphinx can maintain the performance loss of the applications such that the LP value can be met, however in Figure 52(d), Sphinx is unable to maintain the performance loss for Memcached. This is due to a limitation on how much delay the Congestion Control delays can add and their impact on the application latency.

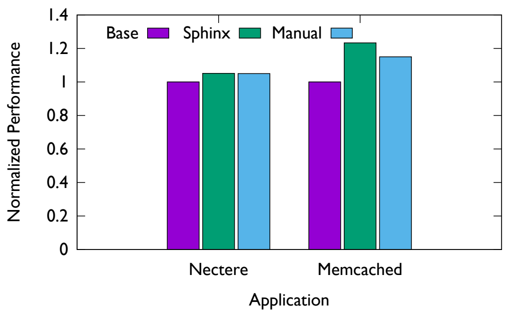


(a) 100.5 vs 99.5us

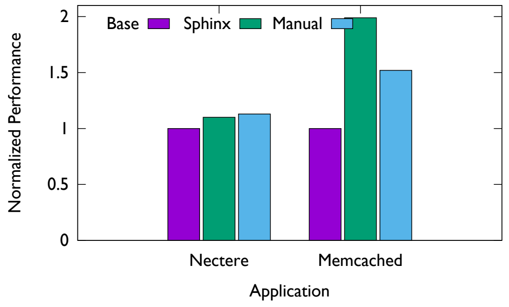


(b) 48us vs 187us

Figure 53: Normalized Application Average Latency when comparing Sphinx and Manual for LP 1:1.



(a) 212us vs 187us



(b) 48us vs 187us

Figure 54: Normalized Application Average Latency when comparing Sphinx and Manual for LP 1:4.

Three, Figures 53 and 54 show that Sphinx can meet the required application latency when compared to the Manual policy.

In Figure 55 we highlight another feature of Sphinx where it allows us to change the target LP dynamically. When running Nectere and Memcached application with target LP 1:4, we change the target to LP 1:1. Sphinx can dynamically adapt the network resources to meet the new target LP by reducing or increasing resources for Memcached based on the monitored latency from the Host Agent. This is possible through our Pricing control loop where the new target LP would trigger a Price decrease which further implies that a resource allocation decision must be made.

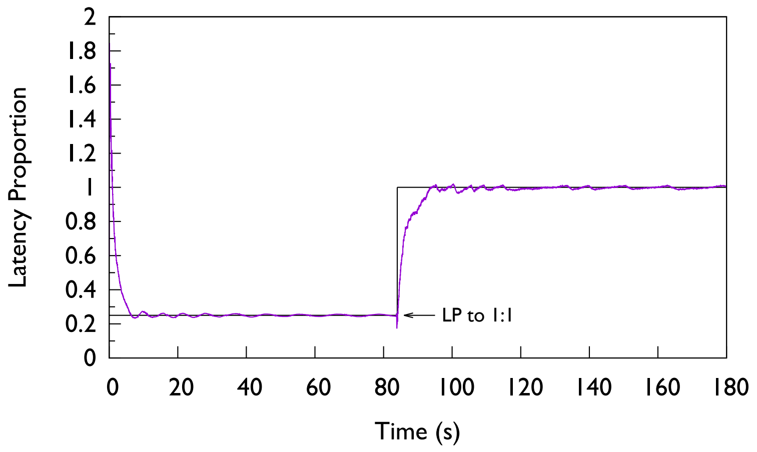


Figure 55: Sphinx allows us to change the LP value dynamically. Nectere (212us) and Memcached (187us) used here.

7.8 Chapter Summary

In this chapter we have shown that there are various challenges in provisioning collocating performance-critical and data-intensive workloads in hardware-virtualized clusters using Software-Defined platforms. We address this problem by defining a new latency-based fairness criteria called 'Latency-Proportionality' that allows the user to specify how performance between such flows should be maintained. We use our Virtual Currency Abstraction to build a control model that continuously monitors the set of heterogeneous flows that will exist on such platforms. We develop, Sphinx, our network management platform that uses these techniques to enable LP for such type of flows. Sphinx, is extensible with respect to the type of flows to be monitored as well as the resource controls that can be used to provision each flow.

We demonstrate the utility of Sphinx to maintain the LP property for latency-sensitive and data-intensive applications within a SR-IOV InfiniBand cluster. Sphinx leverages the network hardware mechanisms as well as host-based controls to limit or increase the network resources allocated to each flow. The limitations in Sphinx arise from its dependence on the coarse-grained IB Congestion Control and QoS mechanisms to reduce interference between such collocated applications for certain workload behaviors. Sphinx is user configurable through REST APIs to specify policies and can dynamically orchestrate the network to meet the policy LP requirement.

CHAPTER VIII

PREVIOUS WORK

In this section we discuss prior research that is related to this thesis. We categorize the research into various areas that thesis most closely identifies with.

8.1 Memory Introspection and Monitoring Network Devices

Memory introspection techniques have been used extensively for monitoring VMs. VMSafe [130] is a security framework provided by VMWare for their industry-standard hypervisor, ESX Server, which includes sets of APIs to perform introspection of different resources of the system. [38] discusses a technique called Virtual Machine Introspection (VMI) used to construct an Intrusion Detection System (IDS). VMI monitors resources like CPU, memory, and network used by the virtual machine, and the IDS detects any attacks originating from the VM. Similarly, Payne et al. [13] develop a framework for virtual memory introspection and virtual disk monitoring for the Xen hypervisor. While IBMon targets InfiniBand devices, as a sample device with VMM-bypass capabilities, other devices with similar capabilities [107, 54, 28] can be monitored via the same techniques. Other hardware-based techniques [60, 108] monitor the VM's memory for detect malicious changes. More recent work, Pingmesh [43] and Planck [115] can monitor high throughput networks out-of-band to get overall application performance.

8.2 Economics and Resource Management

We are not the first ones to use congestion pricing for resource management. There has been earlier work in computer networks for allocations [72], congestion avoidance [66], energy management [93] that also used this model to reduce congestion. Also, it is used in QoS management [96] for providing better CPU scheduling for tasks. Currencies have also been used for providing energy management as shown in [141, 142]. In addition, market-based economic strategies are also used for resource allocation in large datacenters. This is very prevalent in cloud computing infrastructures since these models make it easier to allocate the physical infrastructure [24, 34, 128]. More recent work in [22] has investigated on how to reduce costs for collocated HPC workloads while mitigating interference.

8.3 Virtualized Resource Management in Clusters

8.3.1 Host-based Resource Management

Several research efforts by our own group as well as others focus on dynamic monitoring and analysis on the behavior of applications deployed on a single system or across distributed nodes [3, 127, 31, 99, 102]. Other work provide functionalities such as node-level power management [94], CPU scheduling and memory allocation [62] or VM deployment and migration in cluster settings [27, 70, 139, 80]. There are several industry-related efforts [21, 41, 105] that provide support for network/storage bandwidth guarantees for VMs within a datacenter, however, they are limited to the hypervisor that is used or uses an emulated platform as VM workload. There also is other recent research on modifying vNICs to provide fairness guarantees [11], however this effort relies on modifying specialized hardware and might provide limited functionality. A key distinction in our resource management work is that it enables monitoring of devices and VM interactions with such devices which occur 'outside' of the virtualization layer. To the best of our knowledge, there hasn't been prior work focusing on dynamic monitoring of the VMs or applications' use of devices such as the IB HCAs, when accessed in a manner which bypasses the operating system or the hypervisor layer.

8.3.2 Distributed Network Management

Many recent efforts have explored distributed control for providing network guarantees for cloud-based workloads. These have looked at providing min-max fairness to workloads [104], providing minimum bandwidth guarantees [42], or using congestion notifications from switches [23]. Mogul et. al. in [87] provide a detailed survey of these approaches. There are also other efforts that provide network guarantees for per-tenant [73] and inter-tenant communication [14]. Authors in Gatekeeper [122] enforce limits per tenant per physical machine by providing exact egress and ingress bandwidth values. In other work we similarly enforce such bandwidth values dynamically by providing prices and setting CPU Cap limits or Network Controls. Some recent efforts [57, 29, 5] provide performance isolation for VMs by constructing distributed virtual switches, faster performance and performance guarantees to network flows respectively in the datacenter.

8.3.3 Software-Defined Network Management

As the popularity of open source efforts [82, 88] increases, there is a growing need among cloud infrastructure manufacturers to provide compatible hardware for such platforms [83]. With high performance fabrics it is important to maintain that performance while integrating the NICs with the infrastructure. [12] enables some support for cloud computing applications within a supercomputer infrastructure, however, this effort is limited to the type of supercomputer used, BlueGene/P, and not to more commodity fabrics/clusters available. Other efforts [89, 121] enable different architectures for the network by leveraging such software-defined networks. By providing such support they demonstrate the flexibility and deployability of SDNs. Some vendors have added support on the hardware to newest generation Ethernet devices in order to provide high performance [84]. While this approach is extremely viable, it seems to neglect older generation devices that may not have such support available. One of the areas that these approaches lack/limit is the support for enabling Quality of Service (QoS) and performance isolation between VMs. As more VMs are consolidated using Software Defined Networks, there has to be support enabled at the controller level to provide isolation between groups of VMs that share the same links. Also, such support is usually configured in a static manner which may lead to lower network utilization or performance guarantee misses, thus motivating the need for Sphinx-level dynamic mechanisms to configure these in conjunction with application monitoring techniques.

8.4 Network Management for Low-Latency Workloads

Providing resource management for lower latencies is becoming increasingly common and a number of recent works has tried to address this problem in various ways. In order to provide latency guarantees, various efforts [40, 56, 10] utilize rate limiters built into operating systems or virtual switches to control the injection of packets into the network. While these approaches are geared towards Ethernet-based networks, the core idea of delaying some packets for specific applications is used by us as well for RDMA-based and other workloads. However, as networks move towards 100Gbps, it will be increasing difficult for software rate limiters to keep up with application traffic and such controls must be pushed to hardware [119, 64, 36]. Also, we believe that application programming models like RDMA to provide direct network access will be more common to help applications use the maximum network throughput.

Using hardware-based mechanisms to reduce congestion are also increasingly common. Networks utilize notification mechanisms on the wire between hosts and switches to find whether the ports are overloaded. While, approaches like DCQCN [145] and

Timely [86] generally alleviate congestion in the network overall, Sphinx leverages similar existing mechanisms to ‘delay’ the packets to prevent congestion from occurring in the first place as well as use the application performance to tune the network accordingly.

Chronos [63] is another system that provides an overall approach to reduce end-to-end latencies in all areas of the network stack as well as provide an optimized network controller to minimize application latency. It will be useful for Sphinx to take advantage of such a controller to help further reduce latencies.

There is also recent work in the area of providing bandwidth guarantees for cloud tenants that is related to our work since the fairness and guarantees provided can be viewed as a set of dynamic bandwidth guarantees. Oktopus [15], CloudMirror [75] motivated our flow model design used by applications to specify SLOs. These efforts aim to provide notions of bandwidth fairness, minimum guarantees or maximize network utilization, which can be used within Sphinx as diverse sets of policies.

CHAPTER IX

CONCLUSION

This thesis addresses the problems with managing I/O for VMs within clusters that are connected using high performance fabrics by creating a flexible, dynamic resource management system through hypervisor enlightenment and Economics-based resource allocations. These abstractions allow the collocation of various performance-critical and data-intensive workloads, on single as well as distributed nodes in order to meet their SLO objectives. We present the following contributions that re-assert the objectives outlined in the beginning of the thesis.

IBMon, is our hypervisor-based monitoring tool for RDMA-based workloads and fabrics which allows low-level and fine-grained network monitoring through memory introspection techniques. Using the tool hypervisors are now aware of the network utilization by each VM. This further enables the construction of network management platforms that can more effectively manage such fabrics.

FaReS and Resource Exchange, our example platforms that manage I/O on a per-node basis for collocated bandwidth-intensive and latency-sensitive applications respectively. FaReS presents observations and results that underpin the tradeoffs regarding how such fabrics should be managed. ResEx introduces our Congestion Pricing-based resource management abstraction to control the I/O for latency-sensitive applications. ResEx shows that through such Pricing abstractions VMs can meet their latency guarantees even when there is a high level of interference.

The observations and results from ResEx have allowed us to design and construct Distributed Resource Exchange that extends our Pricing-based resource management to consider workloads deployed on multiple nodes. DRX co-ordinates resource allocations on a cluster-basis by monitoring VM Ensembles and their impact on latency-sensitive workloads when deployed on shared SR-IOV InfiniBand clusters. DRX allows low-latency workloads to meet their SLAs to within 15% by penalizing distributed applications only when they are found to cause 'hurt'.

Finally, we present Sphinx, our software-defined fabric management solution based on the principle of Resource Exchange that allows performance-critical and data-intensive workloads to be collocated in a hardware-virtualized InfiniBand platform. Sphinx introduces a fairness concept called 'Latency Proportionality' that defines what performance degradation is acceptable for these workloads through user-defined policies. By leveraging network hardware mechanisms to control interference we can dynamically orchestrate the network to meet the policy goals for low-latency workloads.

The limitations in our resource management techniques arise due to the control mechanisms available for VMM-bypass I/O to affect the workload I/O performed at a finer granularity. When workload behavior is within certain I/O-intensiveness for the minimal allocation, the mechanisms prescribed here operate to provide the required fairness and performance guarantees.

Overall, the contributions of this thesis enable current hypervisors and network management utilities to monitor such high performance fabrics as well as provide finer-grained allocations to help multiple applications meet their performance objectives when collocated on such clusters. Though the mechanisms described in this thesis are general, they can easily be extended to consider the capabilities and properties of future networks in order to manage them.

CHAPTER X

FUTURE DIRECTIONS

This thesis provides several different aspects to consider for future work. The IBMon tool can be extended with regards to different network capabilities like timestamps. Future RDMA networks will contain a timestamp on each CQE, that would allow IBMon to provide much more accurate latency and bandwidth information for each VM or an application deployed.

Also, future network devices will have per-VF resource controls, i.e. have QoS or CC settings described on a per-VF basis, that allows much more finer grained management. The mechanisms described in Sphinx can easily be extended to leverage these capabilities and can easily provide a better resource management model. These capabilities can further be used to enforce end-to-end SLAs for distributed low-latency workloads by using DRX to co-ordinate network configuration actions across hosts. Also, with future devices like Omnipath [118] allowing control for each application at the network flit-level, better packet scheduling would enable better latency guarantees as well.

The Pricing-based mechanisms allows various Economics-based schemes to be combined with cloud computing model. For example, Revenue and Billing services can use these mechanisms to penalize the cloud provider for missing SLOs of applications or the cloud user can be charged exactly based on the I/O used for VMM-bypass networks. The network management model can be integrated with other system-level resource management techniques like VMware DRS [132], Island-based Management [125] to provide a fully-integrated cloud resource management platform that can provide latency as well as fairness guarantees from all resource perspectives.

REFERENCES

[1] "Direct Data Placement over Reliable Transports." `http://tools.ietf.org/html/rfc5041`.

[2] Abramson, D. and others, "Intel Virtualization Technology for Directed I/O," *Intel Technology Journal*, vol. 10, no. 3, 2006.

[3] Agarwala, S. and Schwan, K., "SysProf: Online Distributed Behavior Diagnosis through Fine-grain System Monitoring," in *ICDCS*, 2006.

[4] Al-Fares, M., Radhakrishnan, S., Barath, R., Nelson, H., and Amin, V., "Hedera: Dynamic Flow Scheduling for Data Center Networks," in *Proceedings of the USENIX Conference on Networked Systems Design and Implementation*, Apr 2010.

[5] Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., and Vahdat, A., "Hedera: Dynamic Flow Scheduling for Data Center Networks," in *Proceedings of NSDI*, 2010.

[6] Alfaro, F. J., Sanchez, S., and Duato, D., "QoS in InfiniBand Subnetworks," *IEEE Transactions on Parallel Distributed Systems*, 2004.

[7] Alizadeh, M., Kabbani, A., Edsall, T., Prabhakar, B., Vahdat, A., and Yasuda, M., "Less is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center," in *Proceedings of NSDI*, 2012.

[8] "Amazon Web Services: Case Studies." http://aws.amazon.com/solutions/case-studies.

[9] "AMD I/O Virtualization Technology." http://tinyurl.com/a6wsdwe.

[10] Angel, S., Ballani, H., Karagiannis, T., O'Shea, G., and Thereska, E., "End-to-end Performance Isolation through Virtual Datacenters," in *OSDI'14: The 11th USENIX Symposium on Operating Systems Design and Implementation*, Oct 2014.

[11] Anwer, M. B., Nayak, A., Feamster, N., and Liu, L., "Network I/O Fairness in Virtual Machines," in *Proceedings of SIGCOMM VISA*, 2010.

[12] Appavoo, J., Waterland, A., Da Silva, D., Uhlig, V., Rosenburg, B., Van Hensbergen, E., Stoess, J., Wisniewski, R., and Steinberg, U., "Providing a Cloud Network Infrastructure on a Supercomputer," in *Proceedings of HPDC*, 2010.

[13] B. Payne and M. Carbone and W. Lee, "Secure and Flexible Monitoring of Virtual Machines," in *Proceedings of ACSAC*, 2007.

[14] Ballani, H., Jhang, K., Karagiannis, T., and et. al., C. K., "Chatty Tenants and the Cloud Network Sharing Problem," in *Proceedings of NSDI*, 2013.

[15] Ballani, Hitesh and Costa, Paolo and Karagiannis Thomas and Rowstron Ant, "Towards Predictable Datacenter Networks," in *Proceedings of the ACM SIGCOMM Conference*, SIGCOMM '11, pp. 242–253, Aug 2011.

[16] Barham, P., Dragovic, B., Fraser, K., Hand, S., and others, "Xen and the Art of Virtualization," in *Proceedings of SOSP*, 2003.

[17] Barker, S. K. and Shenoy, P., "Empirical Evaluation of Latency-sensitive Application Performance in the Cloud," in *Proceedings of MMSys*, February 2010.

[18] Ben-Yehuda, M., Mason, J., Krieger, O., Xenidis, J., Dorn, L. V., Mallick, A., Nakajima, J., and Wahlig, E., "Utilizing IOMMUs for Virtualization in Linux and Xen," in *Ottawa Linux Symposium*, 2006.

[19] Bernt Arne Ødegaard, "C++ programs for Finance." http://finance.bi.no/ bernt/gcc_prog/.

[20] BigSwitchNetworks, "Big Network Controller." http://bigswitch.com/products/SDN-Controller/.

[21] Billaud, J.-P. and Gulati, A., "hClock: Hierarchial QoS for Packet Scheduling in a Hypervisor," in *Proceedings of EuroSys*, apr 2013.

[22] Breslow, A. D., Tiwari, A., Schulz, M., Carrington, L., Tang, L., and Mars, J., "Enabling Fair Pricing on HPC Systems with Node Sharing," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, November 2013.

[23] Briscoe, B. and Sridharan, M., "Network Performance Isolation in Data Centres using Congestion Exposure (ConEx)," 2012. http://datatracker.ietf.org/doc/draft-briscoe-conex-data-centre.

[24] Byde, A., Byde, A., Sallé, M., Sallé, M., Bartolini, C., and Bartolini, C., "Market-Based Resource Allocation for Utility Data Centers," Tech. Rep. HPL-2003-188, HP Labs, sep 2003.

[25] Casado, M., Koponen, T., Ramanathan, R., and Shenker, S., "Virtualizing the Network Forwarding Plane," in *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, nov 2010.

[26] Cherkasova, L. and Gardner, R., "Measuring CPU Overhead for I/O Processing in the Xen Virtual MachineMonitor," in *Proc. of USENIX ATC*, 2005.

[27] Cohen, I., Zhang, S., Goldszmidt, M., Symons, J., Kelly, T., and Fox, A., "Capturing, indexing, clustering and retrieving system history," in *Proc. of SOSP*, 2005.

[28] Consortium, R., "RDMA-enabled NIC." `www.rdmaconsortium.org`.

[29] Curtis, A. R., Mogul, J. C., Tourrilhes, J., Yalagandula, P., Sharma, P., and Banerjee, S., "DevoFlow: Scaling Flow Management for High-Performance Networks," in *Proceedings of SIGCOMM*, 2011.

[30] Dantzig, G. B., "Discrete-variable extremum problems," tech. rep., Rand Corporation, 1957.

[31] Delimitrou, C. and Kozyrakis, C., "Quasar: Resource-efficient and QoS-aware Cluster Management," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2014.

[32] Dong, Y., Yu, Z., and Rose, G., "SR-IOV Networking in Xen: Architecture, Design and Implementation," in *Proceedings of WIOV*, 2008.

[33] Dragojević, A., Narayanan, D., Hodson, O., and Castro, M., "FaRM: Fast Remote Memory," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2014)*, April 2014.

[34] Feldman, M., Lai, K., and Zhang, L., "A Price-Anticipating Resource Allocation Mechanism for Distributed Shared Clusters," in *Proceedings of the ACM Conference on Electronic Commerce*, jun 2005.

[35] Ferdman, M., Adileh, A., Onur, K., Stavros, V., Mohammad, A., Djordje, J., Cansu, K., Daniel, P. A., Anastasia, A., and Babak, F., "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware," in *Proceedings of ASPLOS*, Mar 2012.

[36] Flajslik, M. and Rosenblum, M., "Network Interface Design for Low Latency Request-response Protocols," in *Proceedings of USENIX ATC*, June 2013.

[37] "Project Floodlight." http://www.projectfloodlight.org/floodlight/.

[38] Garfinkel, T. and Rosenblum, M., "A Virtual Machine Introspection Based Architecture for Intrusion Detection," in *Proceedings of NDSS*, 2003.

[39] Govindan, S., Nath, A. R., Das, A., Urgaonkar, B., and Sivasubramaniam, A., "Xen and co.: Communication-Aware CPU Scheduling for Consolidated Xen-based Hosting Platforms," in *Proceedings of VEE*, 2007.

[40] Grosvenor, M. P., Schwarzkopf, M., Ionel, G., N.M., W. R., W., M. A., Steven, H., and Jon, C., "Queues don't matter when you can jump them!," in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2015.

[41] Gulati, A., Merchant, A., and Varman, P. J., "mClock: Handling Throughput Variability for Hypervisor IO Scheduling," in *Proceedings of OSDI*, 2010.

[42] Guo, C., Lu, G., Wang, H. J., and et. al., S. Y., "SecondNet: A Data Center Network Virtualization Architecture with Bandwidth Guarantees," in *Proceedings of ACM CoNext*, 2010.

[43] Guo, C., Yuan, L., Dong, X., Yingnong, D., Ray, H., Dave, M., Zhaoyi, L., Vin, W., Bin, P., Hua, C., Zhi-Wei, L., and Varugis, K., "Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, Aug 2015.

[44] Gupta, D., Cherkasova, L., Gardner, R., and Vahdat, A., "Enforcing Performance Isolation Across Virtual Machines in Xen," in *Proc. of MiddleWare*, 2006.

[45] Gupta, V., Ranadive, A., Gavrilovska, A., and Schwan, K., "Benchmarking Next Generation Hardware Platforms: An Experimental Approach," in *Proceedings of SHAW*, 2012.

[46] "Apache Hadoop." http://hadoop.apache.org/.

[47] Henderson, H. D., "Supply and Demand," 1922.

[48] Himanshu Raj and Karsten Schwan, "High Performance and Scalable I/O Virtualization via Self-Virtualized Devices," in *HPDC*, 2007.

[49] "High Performance Linpack." http://www.netlib.org/benchmark/hpl.

[50] "HP Procurve: Data Center Provisioning Automation." http://www.procurve.com/solutions/enterprise/datacenter/connectivity.htm.

[51] Huang, W., Liu, J., and Panda, D., "A Case for High Performance Computing with Virtual Machines," in *ICS*, 2006.

[52] "InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.2." http://www.infinibandta.org.

[53] Ibaraki, T., "Enumerative approaches to combinatorial optimization," *Annals of Operations Research*, vol. 10, Jan 1988.

[54] InfiniBand Trade Association, "RDMA over Converged Ethernet." `http://www.infinibandta.org/content/pages.php?pg=about_us_RoCE`, 2014.

[55] Islam, N. S., Rahman, M. W., Jose, J., Rajachandrasekar, R., Wang, H., Subramoni, H., Murthy, C., and Panda, D. K., "High Performance RDMA-based Design of HDFS over InfiniBand," in *Proceedings of SC*, November 2012.

[56] Jang, K., Sherry, J., Hitesh, B., and Toby, M., "Silo: Predictable Message Latency in the Cloud," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, Aug 2015.

[57] Jeyakumar, V., Alizadeh, M., Mazières, D., Prabhakar, B., Kim, C., and Greenberg, A., "EyeQ: Practical Network Performance Isolation at the Edge," in *Proceedings of NSDI*, 2013.

[58] Jose, J., Li, M., Lu, X., and et. al., K. C. K., "Sr-iov support for virtualization infiniband clusters: Early experience," in *Proceedings of CCGrid*, apr 2013.

[59] Jose, J. and Subramoni, H. and Miao Luo and Minjia Zhang and Jian Huang and Wasi-ur-Rahman M. and Islam N.S. and Xiangyong Ouyang and Hao Wang and Sur S. and Panda D.K., "Memcached Design on High Performance RDMA Capable Interconnects," in *Proceedings of ICPP*, September 2011.

[60] Jr., N. L. P., Fraser, T., Molina, J., and Arbaugh, W. A., "Copilot -a coprocessor-based kernel runtime integrity monitor," in *Proceedings of the USENIX Security Symposium*, 2004.

[61] Kalia, Anuj and Kaminsky, Michael and Andersen David G., "Using RDMA Efficiently for Key-value Services," in *Proceedings of the ACM Conference on SIGCOMM*, SIGCOMM '14, pp. 295–306, Aug 2014.

[62] Kalyvianaki, E. and Charalambous, T., "On Dynamic Resource Provisioning for Consolidated Servers in Virtualized Data Centers," in *Workshop on Performability Modeling of Computer and Communication Systems*, 2007.

[63] Kapoor, R., Porter, G., Tewari, M., Voelker, G. M., and Vahdat, A., "Chronos: Predictable Low Latency for Data Center Applications," in *Proceedings of SOCC*, 2012.

[64] Kaufmann, A., Peter, S., Anderson, T., and Krishnamurthy, A., "FlexNIC: Rethinking Network DMA," in *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, May 2015.

[65] Kesavan, M., Ranadive, A., Gavrilovska, A., and Schwan, K., "Active CoordinaTion (ACT) - Towards Effectively Managing Virtualized Multicore Clouds," in *Proc. of IEEE Cluster*, 2008.

[66] Key, P., Mcauley, D., Barham, P., and Laevens, K., "Congestion Pricing for Congestion Avoidance," tech. rep., Microsoft Research, 1999.

[67] Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A., "kvm: the Linux Virtual Machine Monitor," in *Proceedings of Linux Symposium*, 2006.

[68] Klasky, S., Ethier, S., Lin, Z., Martins, K., McCune, D., and Samtaney, R., "Grid-Based Parallel Data Streaming Implemented for the Gyrokinetic Toroidal Code," in *Proceedings of the SC*, 2003.

[69] Koponen, T., Amidon, K., Balland, P., Casado, M., Chanda, A., Fulton, B., Ganichev, I., Gross, J., Ingram, P., Jackson, E., Lambeth, A., Lenglet, R., Li, S.-H., Padmanabhan, A., Pettit, J., Pfaff, B., Ramanathan, R., Shenker, S., Shieh, A., Stribling, J., Thakkar, P., Wendlandt, D., Yip, A., and Zhang, R., "Network Virtualization in Multi-tenant Datacenters," in *Proceedings of NSDI*, Apr. 2014.

[70] Kumar, S., Talwar, V., Ranganathan, P., Nathuji, R., and Schwan, K., "M-Channels and M-Brokers: Coordinated Management in Virtualized Systems," in *Workshop on MMCS, HPDC*, 2008.

[71] Kumar, Sanjay and Talwar, Vanish and Kumar, Vibhore and Ranganathan, Parthasarathy and Schwan, Karsten, "vManage: Loosely Coupled Platform and Virtualization Management in Data Centers," in *Proceedings of ICAC '09*, pp. 127–136, 2009.

[72] Kurose, J. F. and Simha, R., "A Microeconomic Approach to Optimal Resource Allocation in Distributed Computer Systems," *IEEE Trans. Comput.*, vol. 38, no. 5, pp. 705–717, 1989.

[73] Lam, T., Radhakrishnan, S., Vahdat, A., and Varghese, G., "NetShare: Virtualizing Data Center Networks across Services," Tech. Rep. CS2010-0957, University of California, San Diego, 2010.

[74] Lee, M., Krishnakumar, A. S., Krishnan, P., Singh, N., and Yajnik, S., "Supporting Soft Real-Time Tasks in the Xen Hypervisor," in *Proceedings of VEE*, 2010.

[75] Lee, Jeongkeun and Turner, Yoshio and Lee Myungjin and Popa Lucian and Banerjee Sujata and Kang Joon-Myung and Sharma Puneet, "Application-driven Bandwidth Guarantees in Datacenters," in *Proceedings of the ACM Conference on SIGCOMM*, SIGCOMM '14, pp. 467–478, August 2014.

[76] "Libvirt: Virtualization api." `http://www.libvirt.org/`.

[77] Liu, J., "Evaluating Standard-Based Self-Virtualizing Devices: A Performance Study on 10 GbE NICs with SR-IOV Support," in *Proceedings of IPDPS*, 2010.

[78] Liu, J., Huang, W., Abali, B., and Panda, D. K., "High Performance VMM-Bypass I/O in Virtual Machines," in *Proc. of USENIX ATC*, 2006.

[79] Marius Hillenbrand and Viktor Mauch and Jan Stoess and Konrad Miller and Frank Bellosa, "Virtual InfiniBand Clusters for HPC Clouds," in *Proceedings of the International Workshop on Cloud Computing Platforms (CloudCP)*, Apr 2012.

[80] Mars, J., Tang, L., Hundt, R., Skadron, K., and Soffa, M. L., "Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, December 2011.

[81] Matthew Andrews, K. K., Ramanan, K., Stolyar, A., Whiting, P., and Vijayakumar, R., "Providing Quality of Service over a Shared Wireless Link," *IEEE Communications*, February 2001.

[82] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J., "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM CCR*, Mar. 2008.

[83] Mellanox Technologies, "Mellanox: Generation of Open Ethernet." http://www.mellanox.com/openethernet/.

[84] Mellanox Technologies, "Mellanox OpenStack and SDN/OpenFlow Solution Reference Architecture." http://www.mellanox.com/sdn/pdf/Mellanox-OpenStack-OpenFlow-Solution.pdf.

[85] Menon, A., Santos, J. R., Turner, Y., Janakiraman, G. J., and Zwaenepoel, W., "Diagnosing Performance Overheads in the Xen Virtual Machine Environment," in *Proceedings of VEE*, 2005.

[86] Mittal, R., Lam, V. T., Nandita, D., Emily, B., Hassan, W., Monia, G., Amin, V., Yaogong, W., David, W., and David, Z., "TIMELY: RTT-based Congestion Control for the Datacenter," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, Aug 2015.

[87] Mogul, J. and Popa, L., "What We Talk About When We Talk About Cloud Network Performance," *ACM CCR*, 2012.

[88] Monsanto, C., Reich, J., Foster, N., Rexford, J., and Walker, D., "Composing Software-Defined Networks," in *Proceedings of NSDI*, 2013.

[89] Mudigonda, J., Yalagandula, P., Mogul, J., Stiekes, B., and Pouffary, Y., "NetLord: A Scalable Multi-Tenant Network Architecture for Virtualized Datacenters," in *Proceedings of SIGCOMM*, 2011.

[90] "Myricom, Inc. Myrinet." www.myri.com.

[91] "Low-Latency 10-Gigabit Ethernet." http://myri.com/Myri-10G/documentation/Low_Latency_Ethernet.pdf.

[92] NASA, "NAS Parallel Benchmarks." http://www.nas.nasa.gov/Resources/Software/npb.htm

[93] Nathuji, R., England, P., Sharma, P., and Singh, A., "Feedback Driven QoS-Aware Power Budgeting for Virtualized Servers," in *Workshop on FeBID*, 2009.

[94] Nathuji, R. and Schwan, K., "VirtualPower: Coordinated Power Management in Virtualized Enterprise Systems," in *Proc. of SOSP*, 2007.

[95] Nathuji, R. and Schwan, K., "Vpm tokens: Virtual Machine-aware Power Budgeting in Datacenters," in *Proceedings of HPDC '08*, pp. 119–128, 2008.

[96] Neugebauer, R. and McAuley, D., "Congestion Prices as Feedback Signals: An Approach to QoS Management," in *ACM SIGOPS Workshop*, 2000.

[97] Nicira, "NOX OpenFlow Controller." `http://www.noxrepo.org`.

[98] Niranjan Mysore, R., Porter, G., and Amin, V., "FasTrak: Enabling Express Lanes in Multi-tenant Data Centers," in *Proceedings of the ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, Dec 2013.

[99] Novaković, D., Vasić, N., Novaković, S., Kostić, D., and Bianchini, R., "DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments," in *Proceedings of USENIX ATC*, June 2013.

[100] "OpenFabrics Enterprise Distribution." http://www.openfabrics.org/.

[101] "OpenFlow - Enabling Innovation in Your Network." http://www.openflow.org/.

[102] Padala, P., Shin, K. G., Zhu, X., Uysal, M., and et al., "Adaptive Control of Virtualized Resources in Utility Computing Environments," in *Proceedings of EuroSys*, 2007.

[103] Pisinger, D., "Core Problems in Knapsack Algorithms," tech. rep., DIKU, University of Copenhagen, Denmark, 1994.

[104] Popa, L., Kumar, G., Chowdhury, M., and et. al., A. K., "FairCloud: Sharing the Network in Cloud Computing," in *Proceedings of ACM SIGCOMM*, 2012.

[105] Popa, L., Yalagandula, P., Banerjee, S., Mogul, J. C., Turner, Y., and Santos, J. R., "ElasticSwitch: Practical Work-Conserving Bandwidth Guarantees for Cloud Computing," *Proceedings of NSDI*, 2013.

[106] Pu, X., Liu, L., Mei, Y., Sivathanu, S., Koh, Y., and Pu, C., "Understanding Performance Interference of I/O Workload in Virtualized Cloud Environment," in *Proceedings of IEEE Cloud*, 2010.

[107] "Quadrics, Ltd. QsNet." www.quadrics.com.

[108] Ram, K. K., Santos, J. R., Turner, Y., Cox, A. L., and Rixner, S., "Achieving 10Gbps using safe and transparent network interface virtualization," in *Proc. of VEE*, 2009.

[109] Ranadive, A. and Davda, B., "Toward a paravirtual vrdma device for vmware esxi guests," tech. rep., VMware, Inc., December 2012.

[110] Ranadive, A., Gavrilovska, A., and Schwan, K., "IBMon: Monitoring VMM-Bypass InfiniBand Devices using Memory Introspection," in *HPCVirtualization Workshop, Eurosys*, 2009.

[111] Ranadive, A., Gavrilovska, A., and Schwan, K., "FaReS: Fair Resource Scheduling for VMM-Bypass InfiniBand Devices," in *Proceedings of CCGrid*, pp. 418–427, 2010.

[112] Ranadive, A., Gavrilovska, A., and Schwan, K., "ResourceExchange: Latency-Aware Scheduling in Virtualized Environments with High Performance Fabrics," in *Proceedings of IEEE Cluster*, 2011.

[113] Ranadive, A., Gavrilovska, A., and Schwan, K., "Distributed Resource Exchange: Virtualized Resource Management for SR-IOV InfiniBand Clusters," in *Proceedings of IEEE Cluster*, 2013.

[114] Ranadive, A., Kesavan, M., Gavrilovska, A., and Schwan, K., "Performance Implications of Virtualizing Multicore Cluster Machines," in *HPCVirtualization Workshop, EuroSys*, 2008.

[115] Rasley, J., Stephens, B., Colin, D., Eric, R., Wes, F., Kanak, A., John, C., and Rodrigo, F., "Planck: Millisecond-scale Monitoring and Control for Commodity Networks," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, Aug 2014.

[116] RedHat Inc., "Redhat resource management guide." `https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/sec-cpu.html`.

[117] Reinemo, S.-A., Gran, E. G., Eimot, M., Skeie, T., Lysne, O., Huse, L. P., and Shainer, G., "First experiences with congestion control in infiniband hardware," in *Proceedings of IPDPS*, March 2010.

[118] Rimmer, T., "Intel Omni-Path Architecture: Enabling Scalable, High Performance Fabrics," in *Proceedings of Symposium on High-Performance Interconnects*, August 2015.

[119] Rumble, S. M., Ongar, D., Stutsman, R., Rosenblum, M., and Ousterhout, J. K., "It's Time for Low Latency," in *Proceedings of HotOS Workshop*, 2011.

[120] Silvano Martello, P. T., *Knapsack Problems: Algorithms and Computer Implementations.* John Wiley, 1990.

[121] Singla, A., Hong, C.-Y., Popa, L., and Godfrey, P. B., "Jellyfish: Networking Data Centers Randomly," in *Proceedings of NSDI*, 2012.

[122] Soares, P. V., Santos, J. R., Tolia, N., and Guedes, D., "Gatekeeper: Distributed Rate Control for Virtualized Datacenters," Tech. Rep. HPL-2010-151, HP Labs, 2010.

[123] Subramoni, H., Petrini, F., Agarwal, V., and Pasetto, D., "Streaming, Low-Latency Communication in On-line Trading Systems," in *Proceedings of IPDPS*, April 2010.

[124] Sugerman, J., Venkitachalam, G., and Lim, B.-H., "Virtualizing I/O Devices on VMware Workstation'sHosted Virtual Machine Monitor," in *Proc. of USENIX ATC*, 2001.

[125] Tembey, P., Gavrilovska, A., and Schwan, K., "Merlin: Application- and Platform-aware Resource Allocation in Consolidated Server Systems," in *Proceedings of the ACM Symposium on Cloud Computing*, November 2014.

[126] "InterContinental Exchange." http://www.theice.com.

[127] Urgaonkar, B. and Shenoy, P., "Sharc: Managing CPU and Network Bandwidth in Shared Clusters," in *IPDPS*, 2004.

[128] "How to Implement Chargeback in a Virtualized Data Center Using the Resource Consumption Model." http://www.vkernel.com/downloads/chargebackmethodology.

[129] "Virtual Machine Device Queues." www.intel.com/network-/connectivity/vtc_vmdq.html.

[130] "VMSafe." www.vmware.com/technology/security/vmsafe.html.

[131] "The VMWare ESX Server." http://www.vmware.com/products/esx/.

[132] VMware, "Vmware distributed resource scheduler, distributed power management." `http://www.vmware.com/products/datacenter-virtualization/vsphere/drs-dpm.html`.

[133] von Eicken, T., Basu, A., Buch, V., and Vogels, W., "U-Net: A User-level Network Interface for Parallel and Distributed Computing," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[134] Wang, C., Rayan, I. A., Eisenhauer, G., Schwan, K., and et. al., "VScope: Middleware for Troubleshooting Time-Sensitive Data Center Applications," in *Proc. of MiddleWare*, 2012.

[135] Wang, Z., Zhu, X., and Singhal, S., “Utilization and SLO-Based Control for Dynamic Sizing of Resource Partitions,” Tech. Rep. HPL-2005-126, Hewlett Packard, 2005.

[136] Wasi-ur Rahman, M., Islam, N. S., Lu, X., Jose, J., Subramoni, H., Wang, H., and Panda, D. K., “High-Performance RDMA-based Design of Hadoop MapReduce over InfiniBand,” in *Proceedings of IPDPSW*, May 2013.

[137] Wei, X., Shi, J., Chen, Y., Chen, R., and Chen, H., “Fast In-memory Transaction Processing Using RDMA and HTM,” in *Proceedings of the 25th Symposium on Operating Systems Principles*, October 2015.

[138] “Congestion Pricing.” http://en.wikipedia.org/wiki/Congestion_pricing.

[139] Wood, T., Shenoy, P., Venkataramani, A., and Yousif, M., “Black-box and Gray-box Strategies for Virtual Machine Migration,” in *NSDI*, 2007.

[140] “Xen Credit Scheduler.” wiki.xensource.com/xenwiki/CreditScheduler.

[141] Zeng, H., Ellis, C. S., Lebeck, A. R., and Vahdat, A., “ECOSystem: managing energy as a first class operating system resource,” *SIGOPS Oper. Syst. Rev.*, October 2002.

[142] Zeng, H., Ellis, C. S., Lebeck, A. R., and Vahdat, A., “Currentcy: A Unifying Abstraction for Expressing Energy Management Policies,” in *Proceedings of the USENIX Annual Technical Conference*, 2003.

[143] “ZeroMQ.” http://zeromq.org.

[144] Zhang, Z., Sun, Z., Chen, H., LV, G., and LU, X., “IBShare: A Dynamic IB Bandwidth Allocation Based on Congestion Control,” *Journal of Information and Computational Science*, vol. 9, no. 18, 2012.

[145] Zhu, Y., Eran, H., Daniel, F., Chuanxiong, G., Marina, L., Yehonatan, L., Jitendra, P., Shachar, R., Haj, Y. M., and Ming, Z., “Congestion Control for Large-Scale RDMA Deployments,” in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, Aug 2015.

VITA

Adit Ranadive was born to Uday Ranadive and Dr. (Mrs.) Nilima Ranadive on February 26th, 1984 in Mumbai (formerly Bombay), a city in the state of Maharashtra in India. He finished his high school in Mumbai and Bachelors in Computer Engineering from Thadomal Shahani Engineering College part of the University of Mumbai in 2006. He completed his Masters in Computer Science in May 2008 from Georgia Institute of Technology and continued his research after enrolling in the Ph.D. program.

He completed his doctoral dissertation in November 2015 after working with Prof. Karsten Schwan and Dr. Ada Gavrilovska. He is working at VMware in Palo Alto, CA and lives there with his wife Smita Vaidya. His primary research interests are in Operating Systems, Virtualization, High Performance Networking, Performance Isolation and general system scalability.