Imperial College London
Department of Computing

Diagnosing, Predicting and Managing Application Performance in Virtualised Multi-Tenant Clouds

Xi Chen

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London
July 2016

Abstract

As the computing industry enters the cloud era, multicore architectures and virtualisation technologies are replacing traditional IT infrastructures for several reasons including reduced infrastructure costs, lower energy consumption and ease of management. Cloud-based software systems are expected to deliver reliable performance under dynamic workloads while efficiently allocating resources. However, with the increasing diversity and sophistication of the environment, managing performance of applications in such environments becomes difficult.

The primary goal of this thesis is to gain insight into performance issues of applications running in clouds. This is achieved by a number of innovations with respect to the *monitoring*, *modelling* and *managing* of virtualised computing systems: (i) Monitoring – we develop a monitoring and resource control platform that, unlike early cloud benchmarking systems, enables service level objectives (SLOs) to be expressed graphically as Performance Trees; these source both live and historical data. (ii) Modelling – we develop stochastic models based on Queueing Networks and Markov chains for predicting the performance of applications in multicore virtualised computing systems. The key feature of our techniques is their ability to characterise performance bottlenecks effectively by modelling both the hypervisor and the hardware. (iii) Managing – through the integration of our benchmarking and modelling techniques with a novel interference-aware prediction model, adaptive on-line reconfiguration and resource control in virtualised environments become lightweight target-specific operations that do not require sophisticated pre-training or micro-benchmarking.

The validation results show that our models are able to predict the expected scalability behaviour of CPU/network intensive applications running on virtualised multicore environments with relative errors of between 8 and 26%. We also show that our performance interference prediction model can capture a broad range of workloads efficiently, achieving an average error of 9% across different applications and setups. We implement this model in a private cloud deployment in our department, and we evaluate it using both synthetic benchmarks and real user applications. We also explore the applicability of our model to both hypervisor reconfiguration and resource scheduling. The hypervisor reconfiguration can improve network throughput by up to 30% while the interference-aware scheduler improves application performance by up to 10% compared to the default CloudStack scheduler.

Acknowledgements

I would like to thank the following people:

- My supervisor, Prof William Knottenbelt, who guided me tirelessly through my PhD. He's a brilliant researcher and advisor who was always ready to help, encourage and push ideas one level further, and share his experience and insight about any problem. His commitment to excellence is inspiring; his belief in his students is motivating; his enthusiasm for research problems is catching. I have been very fortunate to work with him and infected by his knowledge, vision, passion, optimism and inspiration everyday.
- The head of the Department of Computing, Prof Susan Eisenbach, for her support and the financial assistance, the departmental international student scholarship.
- Dr Felipe Franciosi, for his efforts in improving the experiments and ideas, providing insightful knowledge, and encouragement and help in broad areas.
- The DoC Computing Support Group (CSG) for their countless and prompt help, and extremely awesome hand-on knowledge, and incredible problem solving techniques, and their kind, patiences, understanding and endlessly support. In particular: I would like to thank Mr Duncan White, Dr Lloyd Kamara and Thomas Joseph for helping me with the many environmental setups, system configurations and countless difficult problems, and many evenings that they had to work late because of these.
- My collaborators, Chin Pang Ho from Computational Optimisation Group for his great input on the mathematical analysis and many other math problems, and Lukas Rupprecht from LSDS group for his broad knowledge and experience on experiment design, distributed systems and many valuable discussions. The work in this dissertation is the result of collaboration with many other people, more broadly numerous people, Prof Peter Harrison, Giuliano Casale, Tony Field, Rasha Osman, Gareth Jones in AESOP group contributed to the development and refinement of the ideas in this thesis.
- Dr Amani El-Kholy for her endless love, trust, and support for the last 6 years.
- My examiners, Prof Kin Leung and Dr Andrew Rice, for kindly agreeing to serve as internal and external examiners for my viva. Their careful review and their insightful

comments have helped improve the quality of the thesis and have provided directions for my future research.

- And last but not least my family and friends, for their infinite love and perpetual support no matter what I do or what I chose, especially during the hard times that happened 8 hours time difference away from home.

Dedication

To my family

“Just keep swimming.”
Finding Nemo (2003)

Copyright Declaration

© The copyright of this thesis rests with the author and is made available under a Creative Commons Attribution Non-Commercial No Derivatives licence. Researchers are free to copy, distribute or transmit the thesis on the condition that they attribute it, that they do not use it for commercial purposes and that they do not alter, transform or build upon it. For any reuse or redistribution, researchers must make clear to others the licence terms of this work.

Contents

Abstract **i**

Acknowledgements **iii**

1 Introduction **1**

1.1 Motivation 1

1.2 Objectives and Aims 3

1.3 Contributions 5

1.3.1 A Performance Tree-based Monitoring Platform for Clouds 5

1.3.2 Investigating and Modelling the Performance of Web Applications in Multicore Virtualised Environments 7

1.3.3 Diagnosing and Managing Performance Interference in Multi-Tenant Clouds 8

1.4 Statement of Originality and Publications 8

1.5 Thesis Roadmap 9

2 Background **10**

2.1 Introduction 10

2.2 Cloud Computing 10

2.3 Characterising Virtualised Multicore Scalability 11

2.3.1 Multicore and Scalability 12

2.3.2 Linux Kernel Internals and Imbalance of Cores 13

2.3.3 Virtualisation and Hypervisor Overhead 13

2.4 Characterising Performance Interference Caused by Virtualisation 14

2.4.1 Xen and Paravirtualisation 15

2.4.2 CPU Scheduler 17

2.4.3 Disk I/O 19

2.4.4 Grant Table 19

2.4.5 Network I/O 20

2.5 Stochastic Modelling 21

2.5.1 Discrete-time Markov Chains 21

2.5.2 Transition Matrix and State Transition Diagram 23

2.5.3 State Transition Probability Distribution 23

2.5.4 Steady State and Stationary Distribution 26

2.6 Queueing Theory 27

2.6.1 Important Performance Measures 28

2.6.2 Processor-sharing Queueing Models 29

2.6.3 BCMP Network 30

2.7 Performance Trees 32

2.8 Related Research 34

2.8.1 Cloud Benchmarking Systems 34

2.8.2 Performance Modelling of Web Applications in Virtualised Environments 35

2.8.3 Performance Interference Modelling in Multi-Tenant Clouds 38

3 A Performance Tree-based Monitoring Platform for Clouds 42

3.1 Introduction 42

3.2 The Design of a Performance Tree-based Monitoring Platform 43

3.2.1 System Requirements 43

3.2.2 System Architecture 44

3.3 The Myth of Monitoring in Clouds 46

3.3.1 Measuring without Virtualisation 46

3.3.2 Measuring with Virtualisation 48

3.3.3 An Example of Measuring the Actual I/O Queue Size 52

3.4 GUI and Demo in Action 55

3.5 Summary 56

4 Predicting the Performance of Applications in Multicore Virtualised Environments 57

4.1 Introduction 57

4.2 Benchmarking 59

4.3 Proposed Model 63

4.3.1 Model Specification 63

4.3.2 CPU 0 65

4.3.3 Two-class Markov Chain and its Stationary Distribution of CPU 0 65

4.3.4 Average Sojourn Time of CPU 0 68

4.3.5 Average Service Time and Utilisation of CPU 0 70

4.3.6 Likelihood for Estimating Parameters 72

4.3.7 Combined Model . . . 73

4.3.8 Validation . . . 74

4.4 Scalability and Model Enhancement . . . 75

4.4.1 Scalability Enhancement . . . 76

4.4.2 Model Enhancement . . . 76

4.4.3 Prediction with Previous Parameters . . . 77

4.4.4 Prediction validation for Type I hypervisor – Xen . . . 80

4.4.5 Model Limitations . . . 80

4.5 Summary . . . 80

5 Diagnosing and Managing Performance Interference in Multi-Tenant Clouds 82

5.1 Introduction . . . 82

5.2 Characterising Performance Interference . . . 85

5.2.1 Recapping Xen Virtualisation Background . . . 85

5.2.2 Measuring the Effect of Performance Interference . . . 86

5.3 System Design . . . 89

5.3.1 Predicting Performance Interference . . . 91

5.3.2 CPU workloads . . . 92

5.3.3 I/O workloads . . . 93

5.3.4 Virtualisation Slowdown Factor . . . 95

5.4 Interference Conflict Handing . . . 96

5.4.1 Dynamic Interference Scheduling . . . 97

5.4.2 Local Interference Handling . . . 98

5.5 Evaluation 99

5.5.1 Experimental Setup 99

5.5.2 CPU, Disk, and Network Intensive Workloads 100

5.5.3 Mixed Workload 101

5.5.4 MapReduce Workload 102

5.5.5 Interference-aware Scheduling 104

5.5.6 Adaptive Control Domain 107

5.6 Summary 108

6 Conclusion **110**

6.1 Summary of Achievements 111

6.2 Future Work 113

A Xen Validation Results **115**

Bibliography **117**

List of Tables

4.1 Summary of the key parameters in the multicore performance prediction model . 74

4.2 Likelihood estimation of the mean service of class b job 74

4.3 Relative errors between model and measurements (%) 79

5.1 Benchmarking configuration of interference prediction 86

5.2 Specifications for interference-aware scheduler experimental environment 104

5.3 Related work of virtualisation interference prediction 108

List of Figures

1.1 The decision making hierarchy . . . 4

1.2 The overview of this thesis . . . 6

2.1 Context switching inside a multicore server . . . 12

2.2 A brief history of virtualisation . . . 15

2.3 Xen hypervisor architecture with guest virtual machines . . . 17

2.4 A state transition diagram example . . . 24

2.5 An example of Performance Tree query . . . 32

3.1 An example of Performance Tree-based SLO evaluation . . . 44

3.2 System architecture . . . 45

3.3 The screenshot of iostat running fio benchmark tests . . . 48

3.4 The path of an application issuing requests to disks without virtualisation . . . 52

3.5 The sequence diagram of an application issuing requests to disks . . . 53

3.6 The path of an application issuing requests to disks with virtualisation . . . 53

3.7 The sequence diagram of an application issuing requests to disks with virtualisation 54

3.8 Performance Tree evaluation GUI . . . 56

4.1 Testbed Infrastructures for Type-1 Hypervisor (left) and Type-2 Hypervisor (right) 60

4.2 CPU utilisation and software interrupt generated on CPU 0 of 4 core and 8 core VM running web application . . . 62

4.3 Response time and throughput of 1 to 8 core VMs running web application . . . 62

4.4 Modelling a multicore server using a network of queues . . . 63

4.5 State transition diagram of CPU 0 . . . 66

4.6 Comparing numerical and analytical solution $E(K)$ . . . 68

4.7 Response time validation of 1 to 8 core with 1 NIC . . . 75

4.8 Revalidation of response time of 1 to 8 core with multiple number of NICs . . . 78

5.1 The load average (utilisation) example of the VM running sysbench experiment 87

5.2 Co-resident VM performance measurements for CPU, disk and network intensive workloads . . . 88

5.3 CloudScope system architecture . . . 90

5.4 State transition diagrams for CPU, disk and network insensitive workloads . . . 92

5.5 Interference prediction model validation for CPU, disk and network intensive workloads. . . . 100

5.6 Interference prediction and model validation of mixed workloads . . . 102

5.7 Interference prediction model validation for different Hadoop Yarn workloads with different numbers of mappers and reducers . . . 103

5.8 Histograms of the performance improvement of the CloudScope interference-aware scheduler over the CloudStack scheduler . . . 105

5.9 CDF plots for the job completion times of different tasks under CloudScope compared to CloudStack . . . 106

5.10 CloudScope scheduling results compared to the default CloudStack scheduler . . 108

5.11 CloudScope self-adaptive Dom0 with different vCPU weights . . . 108

A.1 Validation of response time of 1 to 8 core with multiple number of NICs on Xen hypervisor . . . 116

Chapter 1

Introduction

1.1 Motivation

Cloud computing refers to both the applications delivered as services over the Internet and the hardware and software in the data centres that provide those services [AFG+10]. The demand for cloud computing has continuously been increasing during recent years. Millions of servers are hosted and utilised in cloud data centres every day and many organisations deploy their own virtualised computing infrastructure [OWZS13, MYM+11]. Virtualisation is the technology that enables the cloud to multiplex different workloads and to achieve elasticity and the illusion of infinite resources. By 2016, it is anticipated that more than 80% of enterprise workloads will be using IaaS (Infrastructure as a Service) clouds [Col16], whether these are public, private and hybrid. Virtualisation and multicore technologies both enable and further encourage this trend, increasing platform utilisation via consolidating multiple application workloads [RTG+12, ABK+14]. This computing paradigm provides improved performance, reduced application design and deployment complexity, elastic handling of dynamic workloads, and lower power consumption compared to traditional IT infrastructures [NSG+13, GNS11a, GLKS11].

Efficient application management over cloud-scale computing clusters is critical for both cloud providers and end users in terms of resource utilisation, application performance and system throughput. Strategies for resource allocation in different applications and virtual resource con-

solidation increasingly depend on understanding the relationship between the required performance of applications and system resources [SSGW11, TZP+16]. To increase resource efficiency and lower operating costs, cloud providers resort to consolidating virtual instances, i.e. packing multiple applications into one physical machine [RTG+12]. Analysis performed by Google shows that up to 19 distinct applications and components are co-deployed on a single multicore node in their data centres [KMHK12]. Understanding the performance of consolidated applications is important for cloud providers to maximise resource utilisation and augment system throughput while maintaining individual application performance targets. Satisfying performance within a certain level of Service Level Objectives (SLOs) is also important to end users because they are keen to know their applications are provisioned with sufficient resources to cope with varying workloads. For instance, local web server infrastructure may not be provisioned for high-volume requests, but it may be feasible to rent capacity from the cloud for the duration of the increasing volume [LZK+11, ALW15].

Further, performance management in clouds is becoming even more challenging with growing cluster sizes and more complex workloads with diverse characteristics. Major cloud service vendors not only provide a variety of VMs that offer different levels of compute power (including GPU), memory, storage, and networking, but also a broad selection of services, e.g. web servers, databases, network-based storage or in-network services such as load balancers [ABK+14, TBO+13]. A growing number of users from different organisations submit jobs to these clouds every day. Amazon EC2 alone has grown from 9 million to 28 million public IP addresses in the past two years [Var16]. The submitted jobs are diverse in nature, with a variety of characteristics regarding the amount of requests, the complexity of processing, the degree of parallelism, and the resource requirements.

Beside the fact that applications running in clouds exhibit a high degree of diversity, they also remain highly variable in performance. Whereas efforts have improved both the raw performance and performance isolation of VMs, the additional virtualisation layer makes it hard to achieve bare metal performance in many cases. Multicore platforms and their current hypervisor-level resource isolation management continues to be challenged in their ability to meet the performance of multiple consolidated workloads [ZTH+13]. This is due to the fact

that an application's performance is determined not only by its use of CPU and memory capacities, which can be carefully allocated and partitioned [BDF+03, HKZ+11], but also its use of other shared resources, which are not easy to control in an isolated fashion, e.g. I/O resources [TGS14]. As a result, cloud applications and appliance throughput varies by up to a factor of 5 times in data centres due to resource contention [BCKR11]. Therefore, developers and cloud managers require techniques to help them measure their applications executed in such environments. Furthermore, they need to be able to diagnose and manage application performance issues.

In summary, resource management in hypervisors or cloud environments must manage individual elastic applications along multiple resource dimensions while dealing with dynamic workloads, and they must do so in a way that considers the runtime resource costs of meeting the application requirements while understanding the performance implications caused by other co-resident applications due to indirect resource dependencies.

1.2 Objectives and Aims

Our primary goal is to develop performance analysis techniques and tools that can help end users and cloud providers to manage application performance efficiently. We tackle this goal in the context of the decision making hierarchy shown in Figure 1.1. At the bottom level, we develop monitoring and resource control tools to help *describe* 'what is going on?' in the system. Next, we *predict* 'what is going to happen if something changes?' using performance models. Finally, at the highest level, we *prescribe* the corresponding performance management actions to answer 'what value-adding decision can we make?' for improved performance.

The primary research hypothesis that emerges from the above is: Within the context of virtualised environments, and with appropriate tool and user interface support, performance modelling can help to diagnose system bottlenecks and to improve resource allocation decision in a way that increases horizontal scalability and lowers interference between co-resident VMs.

Our hypothesis has a broad scope and the related work, such as performance benchmarking

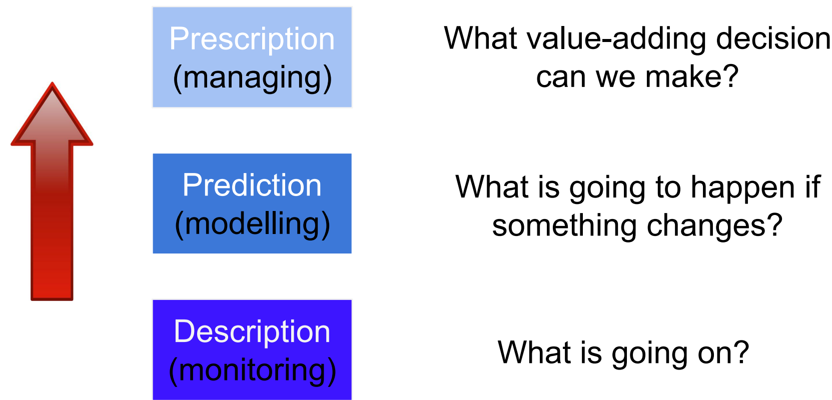


Figure 1.1: The decision making hierarchy

systems, performance modelling and resource management techniques, has been intensively studied. Yahoo!'s YCSB [CST+10], MalStone [BGL+10], and COSBench [ZCW+13] are proposed to create standard benchmarking frameworks for evaluating the performance of different cloud systems, e.g. data processing systems, web services, cloud object storage etc. At the same time, researchers have developed specialised prediction models [BGHK13, CGPS13, BM13, DK13, CH11, NKG10, NBKR13, CKK11] to capture and predict performance under different workloads. These systems and techniques cope with certain aspects of the performance management problem in cloud environments; however, they also come with several drawbacks.

We are going to present the objectives of this dissertation with the observation of current systems and research. In this context the objectives of this thesis are:

1. **To support flexible and intuitive performance queries:** Many benchmarking frameworks provide limited support for flexible and intuitive performance queries. Some of them provide users with a query interface or a textual query language [GBK14], leading to management difficulty for complex performance evaluation and decision making. We seek to design an intuitive graphical specification of complex performance queries that can reason about a broader range of concepts related to application SLOs than current alternatives [DHK04, CCD+01].

2. **To provide an automatic resource control:** If an application does not fulfil the performance requirements, the user must either manually decide on the scaling options

and new instance types, or else write a new module in the system calling corresponding cloud APIs to do so [BCKR11, FSYM13]. Efficient performance management should be enhanced with self-managing capabilities, including self-configuration and self-scaling once the performance is violated.

3. **To provide lightweight and detailed models for targeted prediction:** It is both expensive and unwieldy to compose a prediction model based on comprehensive micro-benchmarks or on-line training [DK13, CH11, NKG10, RKG+13, CSG13, KEY13, ZT12, YHJ+10]. Current clouds require prediction to be general and efficient enough for complex environments where applications change frequently. Apart from being lightweight, low-level resource behaviour such as the utilisation of different CPU cores needs to be captured to support fine-grained and target-specific resource allocation.

4. **To investigate system and VM control strategies for better performance:** Based on the resource demand and performance prediction, successful management needs to be able to not only add or remove servers [SSGW11, ZTH+13, NSG+13], but also to seek the best configuration of the underlying servers and the hypervisor in order to serve the guest VM applications in a more efficient way.

1.3 Contributions

As shown in Figure 1.2, this thesis presents a number of innovations with respect to monitoring, modelling and managing the performance of applications in clouds. We outline the contributions made in this dissertation and several key advantages over current techniques and methods.

1.3.1 A Performance Tree-based Monitoring Platform for Clouds

We implement a real time performance monitoring, evaluation and resource control platform that reflects well the characteristics of contemporary cloud computing (e.g. extensible, user-defined, scalable). The front-end allows for the graphical specification of SLOs using Perfor-

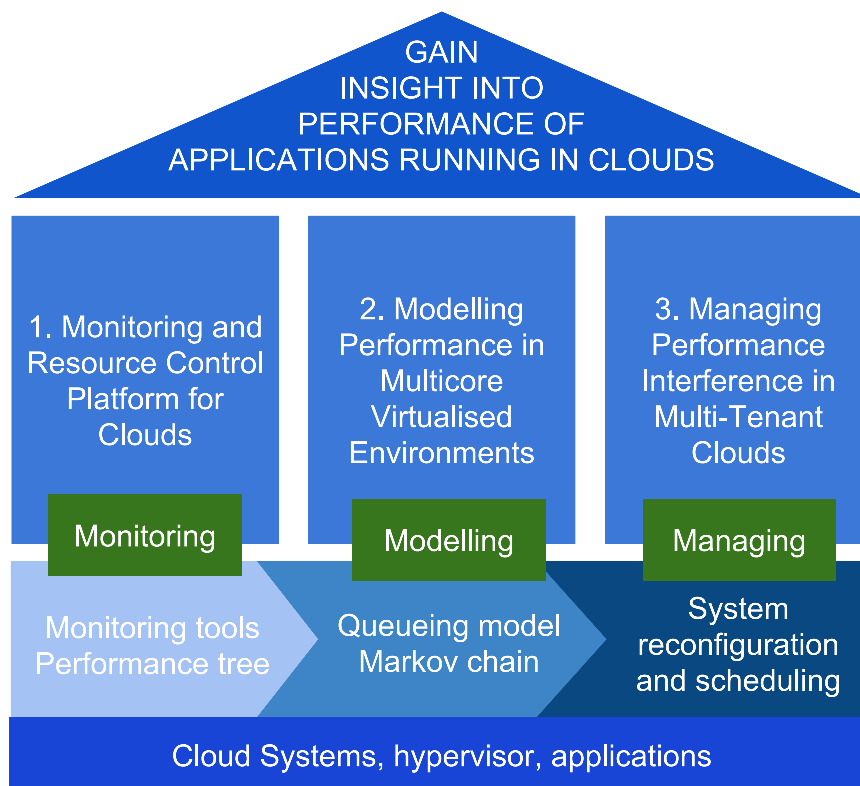


Figure 1.2: The overview of this thesis

mance Trees (PTs), while violated SLOs trigger mitigating resource control actions. SLOs can be specified over both live and historical data, and can be sourced from multiple applications running on multiple clouds. We clarify that our contribution is not in the development of the formalism (which is defined in [SBK06, KDS09]), nor in the Performance Tree evaluation environment [BDK$^+$08]. Rather it concerns the application of Performance Trees to cloud environments. Specifically, we demonstrate how our system is capable of monitoring and evaluating the application performance, and ensuring the SLOs of a target cloud application are achieved by resource auto-scaling. Our system is amenable to multi-tenancy and multi-cloud environments, allowing applications to meet their SLOs in an efficient and responsive fashion through automatic resource control.

1.3.2 Investigating and Modelling the Performance of Web Applications in Multicore Virtualised Environments

As the computing industry enters the Cloud era, multicore architectures and virtualisation technologies are replacing traditional IT infrastructures. However, the complex relationship between applications and system resources in multicore virtualised environments is not well understood. Workloads such as web services and on-line financial applications have the requirement of high performance but benchmark analysis suggests that these applications do not optimally benefit from a higher number of cores [HBB12, HKAC13]. We begin by benchmarking a real web application, noting the systematic imbalance that arises with respect to per-core workload. Having identified the reason for this phenomenon, we propose a queueing model which, when appropriately parametrised, reflects the trend in our benchmark results for up to 8 cores. Key to our approach is providing a fine-grained model which incorporates the idiosyncrasies of the operating system and the multiple CPU cores. Analysis of the model suggests a straightforward way to mitigate the observed bottleneck, which can be practically realised by the deployment of multiple virtual NICs within the VM. It is interesting to add virtual hardware to the existing VM to improve performance (at no actual hardware cost). We validate the model against direct measurements based on a real system. The validation results

show that the model is able to predict the expected performance across different number of cores and virtual NICs with relative errors ranging between 8 and 26%.

1.3.3 Diagnosing and Managing Performance Interference in Multi-Tenant Clouds

Virtual machine consolidation is attractive in cloud computing platforms for several reasons including reduced infrastructure costs, lower energy consumption and ease of management. However, the interference between co-resident workloads caused by virtualisation can violate the SLOs that the cloud platform guarantees. Existing solutions to minimise interference between VMs are mostly based on comprehensive micro-benchmarks or online training which makes them computationally intensive. We develop CloudScope, a system for diagnosing interference for multi-tenant cloud systems in a lightweight way. CloudScope employs a discrete-time Markov chain model for the online prediction of performance interference of co-resident VMs. It uses the results to (re)assign VMs to physical machines and to optimise the hypervisor configuration, e.g. the CPU share it can use, for different workloads. We implement CloudScope on top of the Xen hypervisor and conduct experiments using a set of CPU, disk, and network-intensive workloads and a real system (MapReduce). Our results show that CloudScope interference prediction achieves an average error of 9%. The interference-aware scheduler improves VM performance by up to 10% compared to the default scheduler. In addition, the hypervisor reconfiguration can improve network throughput by up to 30%.

1.4 Statement of Originality and Publications

I declare that this thesis was composed by myself, and that the work that it presents is my own except where otherwise stated.

The following publications arose from the work carried out during my PhD.:

- **6th ACM/SPEC International Conference on Performance Engineering** (ICPE 2015) [CK15] presents a Performance Tree-based monitoring and resource control platform for clouds. The work presented in Chapter 3 is based on this paper.

- **5th ACM/SPEC International Conference on Performance Engineering** (ICPE 2014) [CHO+14] presents a performance model for web applications deployed in multicore virtualised environments. The work presented in Chapter 4 is based on this paper.

- **23rd International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems** (MASCOTS 2015) [CRO+15] presents a comprehensive system, CloudScope, to predict resource interference in virtualised environments, and use the prediction to enhance the operation of cloud platforms. The work presented in Chapter 5 is based on this paper.

1.5 Thesis Roadmap

The reminder of this dissertation is structured as follows. We explore the background and the related work in Chapter 2. In Chapter 3, we introduce a Performance Tree-based monitoring and resource control platform. In Chapter 4, we present a model that captures the performance of web applications in multicore virtualised environments. In Chapter 5, we present CloudScope, a system that diagnoses the bottlenecks of co-resident VMs and mitigates their interference based on a lightweight prediction model. Chapter 6 concludes this dissertation with a summary of our achievements and a discussion of future work.

Chapter 2

Background

2.1 Introduction

This chapter presents background related to the understanding and modelling of the systems that we focus on. We begin by giving an overview of cloud computing, followed by a description of two major enabling technologies heavily used in clouds: multicore architectures and virtualisation. We highlight the most important properties of the workloads and systems with a view to incorporate them into an analytical model. We then present an overview of Markov chains and Queueing theory, which are the key techniques we use for performance modelling. Next, we briefly recap how Performance Trees can be utilised for graphical performance queries and evaluation. We conclude by reviewing the related scientific literature covering cloud benchmarking systems, performance modelling for virtualised multicore systems and performance interference prediction and management in multi-tenant cloud environments.

2.2 Cloud Computing

Cloud computing platforms are becoming increasingly popular for hosting enterprise applications due to their ability to support dynamic provisioning of virtualised resources. Multicore

systems are widespread in all types of computing systems, from embedded to high-performance servers, which are widely provided in these public cloud platforms. More than 80% of mail services (e.g. Gmail, or Yahoo!), personal data storage (e.g. Dropbox), on-line delivery (e.g. Domino pizza), video streaming services (e.g. Netflix) and many other services are supported by cloud-based web applications [CDM+12, ABK+14, ALW15, PLH+15]. Flexibility, high scalability and low-cost delivery of services are key drivers of operational efficiency in many organisations. However, the sophistication of the deployed hardware and software architectures makes the performance studies of such applications very complex [CGPS13]. Often, enterprise applications experience dynamic workloads and unpredictable performance [ABG15, ZCM11]. As a result, provisioning appropriate and budget-efficient resource capacity for these applications remains an important and challenging problem.

Another reason for variable performance and a major feature of cloud infrastructure, is the fact that the underlying physical infrastructure is shared by multiple virtual instances. Although modern virtualisation technology provides performance isolation to a certain extent, the combined effects from concurrent applications, when deployed on shared physical resources, are difficult to predict, and so it is problematic to achieve application SLOs. For example, a MapReduce cloud service has to deal with the interference deriving from contention in numerous hardware components, including CPU, memory, I/O bandwidth, the hypervisor, and their joint effects [BRX13, DK13, WZY+13]. Our goal is to build on the capability of performance modelling to provide significant gains in automatic system management, in particular (a) improving the horizontal scalability of a multicore virtualised system and (b) diminishing the interference between co-resident VMs. We discuss the key aspects of each of these objectives in the subsections below.

2.3 Characterising Virtualised Multicore Scalability

Here we present the background related to capturing the behaviour under scaling of CPU and network intensive workloads (e.g. web applications) running on multicore virtualised platforms.

We start by briefly introducing multicore architectures; then we explain the basic steps involved in receiving/transmitting traffic from/to the network and finally discuss the overhead introduced by virtualisation.

2.3.1 Multicore and Scalability

To exploit the benefits of a multicore architecture, applications need to be parallelised [PBYC13, VF07]. Parallelism is mainly used by operating systems at the process level to provide multitasking [GK06]. We assume that the following two factors are inherent to web applications which scale with the number of cores: (1) the workload of a web application typically involves multiple concurrent client requests on the server and hence **is easily parallelisable**; (2) they exploit the multithreading and asynchronous request services provided by modern web servers (such as Nginx). Each request is usually processed in a separate *thread* and threads can run simultaneously on different CPUs. As a result, modern web servers can efficiently utilise multiple CPU cores. However, scalability of web servers is not linear in practice as other factors, such as synchronisation, sequential workflows, communication overhead, cache pollution, and call-stack depth [NBKR13, VF07, CSA+14, JJ09] limit the performance.

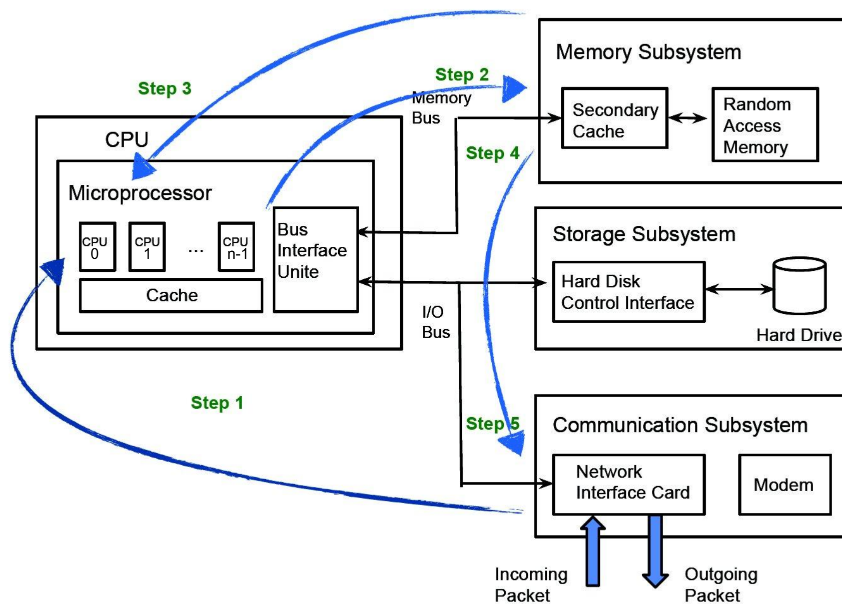


Figure 2.1: Context switching inside a multicore server

2.3.2 Linux Kernel Internals and Imbalance of Cores

Modern computer architectures are interrupt-driven. If a device, such as a network interface card (NIC) or a hard disk, requires CPU cycles to support an I/O operation, it triggers an interrupt which calls a corresponding handler function [TB14]. As we investigate web applications, we focus on **interrupts generated by NICs**. When packets from the network arrive, the NIC places these in an internal packet queue and generates an interrupt to notify the CPU to process the packet. By default, an interrupt is handled by a single CPU (usually *CPU 0*). Figure 2.1 illustrates the process of passing a packet from the network to the application and sending a response back to the network (steps 1 to 5). The NIC driver copies the packet to memory and generates a *hardware interrupt* to signal the kernel that a new packet is readable (step 1). A previously registered interrupt handler is called which generates a *software interrupt* to push the packet down to the appropriate protocol stack layer or application (step 2) [WCB07]. By default, a NIC software interrupt is handled by *CPU 0 (core 0)* which induces a non-negligible load and, as processing rates increase, creates a major bottleneck for web applications (*interrupt storm*) [VF07]. Modern NICs can support multiple packet queues. The driver for these NICs typically provides a kernel module parameter to configure multiple queues [PJD04, JJ09] and assign an interrupt to the corresponding CPU (step 3). The signalling path for PCI (Peripheral Component Interconnect) devices uses message signalled interrupts (MSI-X) that can route each interrupt to a particular CPU [PJD04, HKAC13] (step 4). The NIC triggers an associated interrupt to notify a CPU when new packets arrive or depart on the given queue (step 5).

2.3.3 Virtualisation and Hypervisor Overhead

In the context of modelling the performance of web applications running in virtualised environments, the relationship between application performance and virtualisation overhead must be taken into account. **The virtualisation overhead greatly depends on the different requirements of the guest workloads on the host hardware**. With technologies like

VT-x/AMD-V[1] and nested paging[2], the CPU-intensive guest code can run very close to 100% native speed whereas I/O can take considerably longer due to virtualisation [GCGV06]. For example, Barham et al. [BDF+03] show that the CPU-intensive SPECweb99 benchmark and the I/O-intensive Open Source Database Benchmark suite (OSDB) exhibit very different behaviour in native Linux and XenoLinux (based on the Xen hypervisor). PostgreSQL in OSDB places considerable load on the disk resulting in multiple domain transitions which are reflected in the substantial virtualisation overhead. SPECweb99, on the other hand, does not require these transitions and hence obtain very nearly the same performance as a bare machine [BDF+03].

2.4 Characterising Performance Interference Caused by Virtualisation

The IT industry's focus on virtualisation technology has increased considerably in the past few years. In this section, we will recap the history of virtualisation and the key technologies behind it. Figure 2.2 shows a brief overview of virtualisation technology over time. The concept of virtualisation is generally believed to have its origins in the mainframe days back in the late 1960s and early 1970s when IBM invested a lot of time and efforts in developing robust time-sharing solutions[3], e.g. the IBM CP-40 and the IBM System/370. Time-sharing refers to the shared usage of computer resources among a large group of users, aiming to increase the efficiency of both the user and the expensive compute resources they share. The best way to improve resource utilisation, and at the same time simplify data centre management, is through this time-sharing idea – virtualisation. Virtualisation is defined as enabling multiple operating systems to run on a single host computer, and the essential component in the virtualisation stack is the hypervisor. Hypervisors are commonly classified as one of these two types:

[1] "Hardware-assisted virtualization," in *Wikipedia: The Free Encyclopedia*; available from `https://en.wikipedia.org/wiki/Hardware-assisted_virtualization`; retrieved 9 June 2016.

[2] "Second Level Address Translation," in *Wikipedia: The Free Encyclopedia*; available from `https://en.wikipedia.org/wiki/Second_Level_Address_Translation`; retrieved 7 June 2016.

[3] "Brief history of virtualisation," in Oracle Help Center; available from `https://docs.oracle.com/cd/E35328_01/E35332/html/vmusg-virtualization.html`, retrieved 28 May 2016

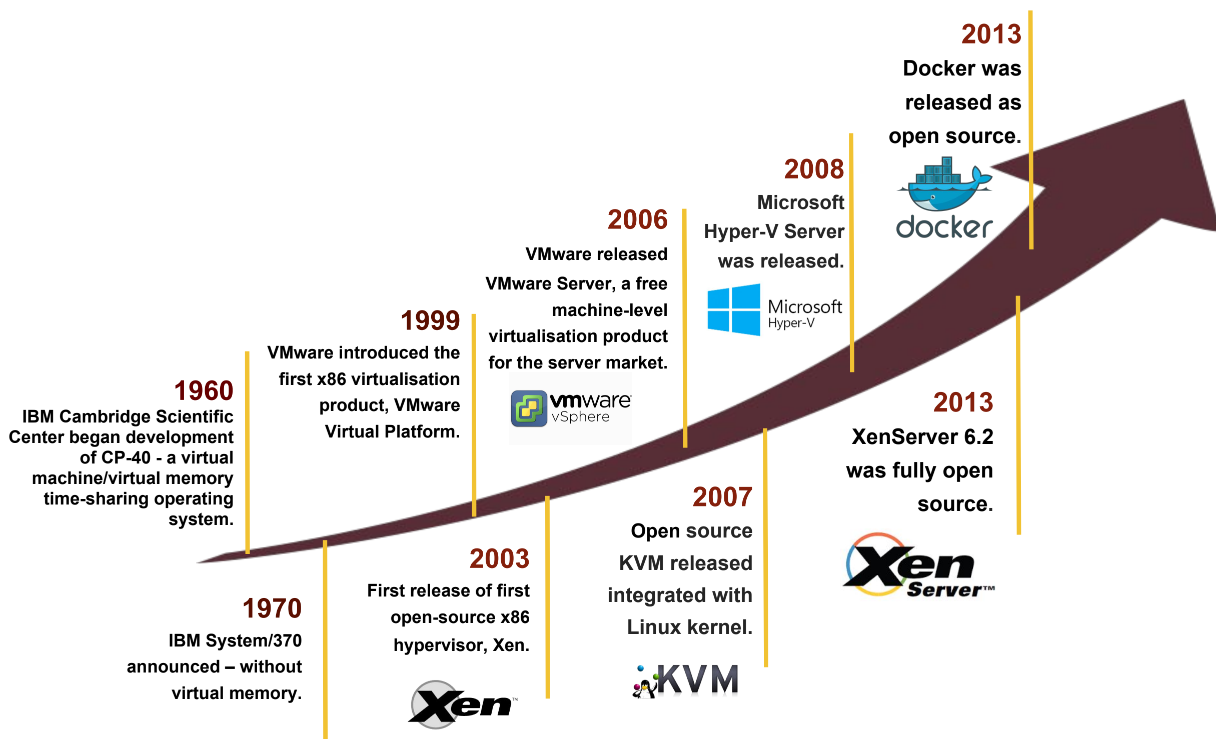


Figure 2.2: A brief history of virtualisation, compiled using information sourced from Timeline of Virtualisation Development[4]

- Type-1 hypervisors run on the host hardware (native or bare metal), and include Xen, Oracle VM, Microsoft Hyper-V, VMware ESX [BDF+03, CFF14, AM13].
- Type-2 hypervisors run within a traditional operating system (hosted), e.g. VMware Server, Microsoft Virtual PC, KVM and Oracle Virtualbox [LWJ+13, TIIN10].

2.4.1 Xen and Paravirtualisation

Most of the Type-2 hypervisors discussed above are well-known commercial implementations of full virtualisation. This type of hypervisor can be considered as a highly sophisticated emulator, which relies on binary translation to trap and virtualise the execution. The hypervisor will allocate a lot of memory to emulate the RAM of the VM and then start reading the disk image of the VM from the beginning (the boot code). There is large overhead in interpreting the VM's instruction from the VM's image and deciding what to do with it, e.g. placing data in

[4] "Timeline of Virtualisation Development," in *Wikipedia: The Free Encyclopedia*; available from `https://en.wikipedia.org/wiki/Timeline_of_virtualization_development`; retrieved 7 June 2016.

memory, drawing on a screen or sending data over the network. This approach can suffer a large performance overhead compared to the VM running natively on hardware. As a result, cloud vendors use Type-1 hypervisors to provide virtual servers. Xen is widely used as the primary VM hypervisor of the product offerings of many public and private clouds. It is deployed in the largest cloud platforms in production such as Amazon EC2 and Rackspace Cloud. We picked Xen as the target of this study in this dissertation due to its popularity and also because it is the only open source bare-metal hypervisor.

Xen provides a spectrum of virtualisation modes, where *paravirtualisation* (PV) and full virtualisation, or hardware-assisted virtualisation (HVM), are the poles. The main difference between PV and HVM mode is that, a paravirtualised guest "knows" it is being virtualised on Xen, contrary to a fully-virtualised guest. To boost performance, fully-virtualised guests can use special paravirtualised device drivers, and this is usually called PVHVM or PV-on-HVM drivers mode. Xen project 4.5 introduced PVH for Dom0 (PVH mode). This is essentially a PV guest using PV drivers for boot and I/O[5].

The entire Xen kernel has been modified to replace privileged instructions with *hypercalls*, i.e. direct calls to the hypervisor. A domain is one of the VMs that run on the system. A hypercall is a software trap from a domain to the hypervisor, just as a system call is a software trap from an application to the kernel[6]. So, instead of issuing a system call to, for example, allocate a memory address space for a process, the PV guest will make a hypercall directly to Xen. This procedure is detailed in Figure 2.3. Domains will use hypercalls to request privileged operations. Like a system call, the hypervisor communicates with guest domains using an event channel. An event channel is a queue of synchronous hypervisor notifications for the events that need to notify the native hardware.

The reason for this mechanism is that, before hardware-assisted virtualisation was invented, the CPU would consider the guest kernel to be an application and silently fail privileged calls, crashing the guest. With hardware support and the drivers in Dom0, we can run the guest as a paravirtualised machine which allows the guest kernel to make the privileged call. The

[5]Xen Project Software Overview. `http://wiki.xen.org/wiki/Xen_Project_Software_Overview`
[6]Hypercall. `http://wiki.xenproject.org/wiki/Hypercall`

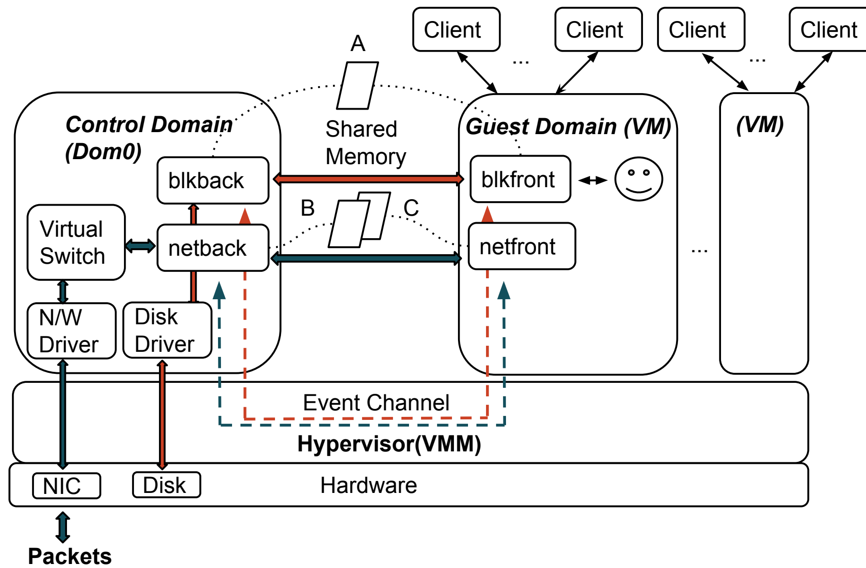


Figure 2.3: Xen hypervisor architecture with guest VMs

hardware will understand the call comes from a guest kernel (the guest domain or DomU) and trap it. It will then pass it to Xen, which in turn will invoke the same hypercall to, for example, update the page table or issue an I/O request [BDF+03] as shown in Figure 2.3. This avoids having to translate general system calls into specific calls to the hypervisor as in the HVM case. PV is very efficient and lightweight compared to HVM and hence it is more widely adopted as a virtualisation solution [XBNJ13, NKG10]. This is why we focus on modelling Xen's PV mode.

We continue our discussion of the Xen internals with the credit-based CPU scheduler, followed by the grant table mechanism, and give more detail about the disk and network I/O data paths in Xen.

2.4.2 CPU Scheduler

When a VM is launched, the control domain will tell the hypervisor to allocate memory and copy the VM's boot code and kernel to that memory. From this point, the VM can run on its own, scheduled directly by the hypervisor. The Xen hypervisor supports three different CPU schedulers which are (1) borrowed virtual time, (2) simple earliest deadline first, and (3)

credit scheduler. All of these allow users to specify CPU allocation via CPU shares (i.e. weights and caps). Credit scheduler is the most widely deployed Xen CPU scheduler after Xen 3.0 was released, because it provides better load balancing across multiple processors compared to other schedulers [CG05, CGV07, PLM+13].

The credit scheduler is a proportional fair-share CPU scheduler designed for SMP hosts[7]. The 'credit' is a combination effect of weight, cap and time slice. A relative weight and a cap are assigned to each VM. A domain with a weight of 512 will get twice as much CPU as a domain with a weight of 256 on the same host. The cap optionally fixes the maximum proportion of CPU a domain will be able to consume, even when the host is idle. Each VM is given 30 ms before being preempted to run another VM. After this 30 ms, the credits of all runnable VMs are recalculated. All VMs in the CPU queues are served as FIFO and the execution of these virtual CPUs (*vCPUs*) ends when the VM has consumed its time slice. The credit scheduler automatically load balances guest vCPUs across all available physical CPUs (*pCPUs*); however, when guest domains run many concurrent workloads, **a large amount of time in the scheduler is spent switching back and forth between tasks instead of doing the actual work**. This causes significant performance overhead.

A VM can enter a boost state which gives the VM a higher priority to be inserted into the head of the queue in case it receives frequent interrupts. These effects of the credit scheduler should be taken into consideration when analysing the performance of co-resident applications, as it is more likely to prioritise an I/O-intensive workload while unfairly penalising CPU-intensive workloads with higher processing latencies. For example, we also observe that executing a CPU-intensive workload within a VM alongside a network-intensive benchmark will result in better throughput for the network-intensive workload. However, to keep our model simple, this work does not consider this effect. We consider CPU-intensive and I/O-intensive (including disk and network) applications are allocated an equal share of CPU resources. We acknowledge this might account for part of the prediction errors observed in Section 5.5.

[7]Credit Scheduler. `http://wiki.xen.org/wiki/Credit_Scheduler`

2.4.3 Disk I/O

While CPU-intensive applications generally exhibit excellent performance (close to bare metal) on benchmarks, applications involving I/O (either storage or network) differ. When a user application inside the guest VM makes a request to a virtual disk, almost every process in the kernel behaves as in the case of a bare metal machine: the system call is processed; a corresponding request for a block device is created; the request is placed in the request queue and the device driver is notified to process the queue. From there, the driver passes the request to the storage media such as a locally attached disk (e.g. SCSI, SATA) or a network-based storage (e.g. NFS, iSCSI). Within a hypervisor, however, the device driver is handled by a module called `blkfront` (see Figure 2.3).

`blkfront` has a shared memory page with another component called `blkback` in the control domain. This page is used to place the requests that come to the request queue of the virtual device. Once the virtual request is placed, `blkfront` notifies `blkback` through an interrupt sent via the event channel. `blkback` will process the request that comes from `blkfront` and make another call to Dom0's kernel. This, in turn, creates another I/O request for the actual device on which the virtual disk allocates. As the virtual disk for a VM can be in various locations, Dom0's kernel is responsible for directing requests coming from `blkback` to the correct place. All these processes add processing overhead and increase the latency for serving I/O requests. **So the actual speed of I/O requests is dependent on the speed and availability of the CPUs of Dom0**. If there are too many VMs performing intensive CPU workloads, the storage I/O data path can be significantly affected [PLM+13, SWWL14].

2.4.4 Grant Table

After a virtual device has been constructed, the guest VM shares a memory area with the control domain which it uses for controlling the I/O requests, the *grant table*[8]. The paravirtualised protocol has a limitation on the amount of requests that can be placed on this shared page

[8]Grant table. `http://wiki.xen.org/wiki/Grant_Table`

between `blkfront` and `blkback` (see Figure 2.3). Furthermore, these requests also have a limitation on the amount of data they can address (the number of segments per request). These two factors lead to **higher latency and lower bandwidth**, which limits I/O performance. The guest will also issue an interrupt to the control domain which goes via the hypervisor; this process has some latency associated with it.

2.4.5 Network I/O

Xen has a database on the control domain which is visible to all VMs through the hypervisor. This is called *XenStore*[9]. When the control domain initialises a VM, it writes certain entries into this database exposing virtual devices (network/storage). The VM network driver will read those entries and learn about the devices it is supposed to emulate. This is the same process as when a physical network card driver loads and examines the available hardware (e.g. PCI bus) to detect which interfaces are available. The Xen VM network driver, called `netfront`, will then communicate directly with another piece of software on the control domain, `netback`, and handshake the network interface.

In contrast to storage, there are **two memory pages** shared between `netfront` and `netback`, each of which contains a circular buffer. One of these is used for packets going out of the guest operating system (OS) and another is used for packets going in to the guest OS from the control domain. When one of the domains (either the guest or the control) has a network packet, it will place information about this request in the buffer and notify the other end via a hypervisor event channel. The receiving module, either `netfront` or `netback` is then capable of reading this buffer and notifying the originator of the message that it has been processed. This is different to storage virtualisation in that storage requests always start in the guest (the drive never sends data without first being asked to do so); therefore, only one buffer is necessary between the control domain and guest domains.

Having presented a brief overview of multicore and virtualisation technology, in the next part of this chapter, we will introduce the theoretical background of this dissertation.

[9] XenStore. `http://wiki.xen.org/wiki/XenStore`

2.5 Stochastic Modelling

To abstract a sequence of random events, one can model it as a stochastic process. Stochastic models play a major role in gaining insight into many areas of computer and natural sciences [PN16]. Stochastic modelling concerns the use of probability to model dynamic real-world situations. Since uncertainty is pervasive, this means that this modelling technique can potentially prove useful in almost all facets of real systems. However, the use of a stochastic model does not imply the assumption that the system under consideration actually behaves randomly. For example, the behaviour of an individual might appear random, but an interview may reveal a set of preferences under which that person's behaviour is then shown to be entirely predictable. The use of a stochastic model reflects only a practical decision on the part of the modeller that such a model represents the best currently available understanding of the phenomenon under consideration, given the data that is available and the universe of models known to the modeller [TK14].

We can describe the basic steps of a stochastic modelling procedure as follows:

1. Identify the critical features that jointly characterise a state of the system under study. This set of features forms the state vector.
2. Given some initial states, understand the universe of transitions which occur between states, together with the corresponding probabilities and/or rates. The set of all states that can be reached from the initial state is known as the reachable state space.
3. Analyse the model to extract the performance measurements of interests, for example, the long run proportion of time spent in end state, or the passage time distribution from one set of state to another.

2.5.1 Discrete-time Markov Chains

Consider a discrete-time random process $\{X_m, m = 0, 1, 2, \dots\}$. In the very simple case where the X_m's are independent, the analysis of this process is relatively straightforward. In particular,

there is no memory in the system so each X_m can be considered independently from previous ones. However, for many real-life applications, the independence assumption is not usually valid. Therefore, we need to develop models where the value of X_m depends on the previous values. In a Markov process, X_{m+1} depends on X_m, but not any the other previous values $X_0, X_1, \dots, X_{m-1}$. A Markov process can be defined as a stochastic process that has this property which is known as Markov property [PN16].

Markov processes are usually used to model the evolution of "states" in probabilistic systems. More specifically, consider a system with a state space $S = \{s_1, s_2, \dots\}$. If $X_n = i$, we say that the system is in state i at time n. For example, suppose that we model a queue in a bank. The state of the system can be defined as the non-negative integer by number of people in the queue. This is, if X_n denotes the number of people in the queue at time n, then $X_n \in S = \{0, 1, 2, \dots\}$. A *discrete-time Markov chain* is a Markov process whose state space S is a finite or countable set, and whose time index set is $T = \{0, 1, 2, \dots\}$ [PN16, TK14]. In formal terms, Discrete-time Markov chains satisfy the Markov property:

$$P(X_{m+1} = j \mid X_m = i, X_{m-1} = i_{m-1}, \dots, X_0 = i_0) = P(X_{m+1} = j \mid X_m = i),$$

for all $t \in \{0, 1, 2, \dots\}$, where $P(X_{m+1} = j \mid X_m = i)$ are the transition probabilities. This notion emphasises that in general transition probabilities are functions not only of the initial and final sates, but also of the time of transition. When the one-step transition probabilities are independent of the time variable m, the Markov chain is said to be *time-homogeneous* [TK14] and $P(X_{m+1} = j \mid X_m = i) = p_{ij}$. In this dissertation, we limit our discussion to this case. As it is common in performance modelling Markov chains are frequently assumed to be time-homogeneous. When time-homogeneous, the Markov chain can be interpreted as a state machine assigning a probability of hopping from each state to an adjacent one, so that the process can be described by a single, time-independent matrix $\mathbf{P} = (p_{ij})$, which will be introduced in the next section. It greatly increases the trackability of the underlying system. Also, it is easy to parametrise the model given the read measurement data.

2.5.2 Transition Matrix and State Transition Diagram

The matrix of one-step transition probabilities is given by $\mathbf{P} = (p_{ij})$. For r states, we have,

$$\mathbf{P} = \begin{bmatrix} p_{11} & p_{12} & \dots & p_{1r} \\ p_{21} & p_{22} & \dots & p_{2r} \\ . & . & . & . \\ . & . & . & . \\ p_{r1} & p_{r2} & \dots & p_{rr} \end{bmatrix}$$

where $p_{ij} \geq 0$, and, for all i,

$$\sum_{k=1}^{r} p_{ik} = 1$$

Consider an example with four states and the following transition matrix:

$$\mathbf{P} = \begin{bmatrix} \frac{1}{10} & \frac{2}{5} & 0 & \frac{1}{2} \\ \frac{2}{5} & \frac{1}{5} & \frac{1}{10} & \frac{3}{10} \\ 0 & \frac{4}{5} & 0 & \frac{1}{5} \\ \frac{1}{5} & 0 & \frac{1}{5} & \frac{3}{5} \end{bmatrix}$$

Figure 2.4 shows the state transition diagram for the above Markov chain. The arrows from each state to other states show the transition probabilities p_{ij}. If there is no arrow from state i to state j, it means that $p_{ij} = 0$.

2.5.3 State Transition Probability Distribution

A Markov chain is completely defined once its transition probability matrix and initial state X_0 are specified (or, more generally, an initial distribution is given a row vector). Recall the Markov chain $\{X_m, t = 0, 1, 2, \dots\}$, where $X_m \in S = \{1, 2, \dots, r\}$. Suppose that the initial

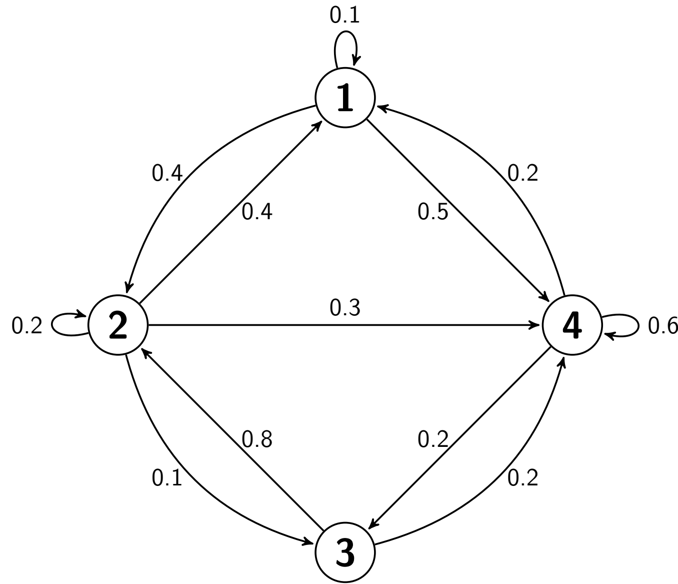


Figure 2.4: A state transition diagram example

distribution of X_0 is known. We can define the row vector $\pi^{(0)}$ as,

$$\pi^{(0)} = \big[P(X_0 = 1), \quad P(X_0 = 2), \quad \dots \quad , P(X_0 = r)\big].$$

We can obtain the probability distribution of X_1, X_2, ..., at time step-n using the law of total probability. More specifically, for any $j \in S$, we have,

$$P(X_1 = j) = \sum_{k=1}^{r} P(X_1 = j \mid X_0 = k) P(X_0 = k) = \sum_{k=1}^{r} p_{kj} P(X_0 = k)$$

If we define

$$\pi^{(m)} = \big[P(X_m = 1), \quad P(X_m = 2), \quad \dots \quad , P(X_m = r)\big],$$

we can obtain the result,

$$\pi^{(1)} = \pi^{(0)}\mathbf{P},$$

Similarly,

$$\pi^{(2)} = \pi^{(1)}\mathbf{P} = \pi^{(1)}\mathbf{P}^2.$$

The n-step transition probability of a Markov chain is,

$$p_{ij}^{(n)} = P(X_{m+n} = j \mid X_m = i) \tag{2.1}$$

and the associated n-step transition matrix is

$$\mathbf{P}^{(n)} = (p_{ij}^{(n)}) \quad (\mathbf{P}^{(1)} = \mathbf{P}).$$

The Markov property allows to express Equation 2.1 in the following theorem.

Theorem 2.1. *The n-step transition probabilities of a Markov chain satisfy*

$$p_{ij}^{(n)} = \sum_{k=0}^{\infty} p_{ik} p_{kj}^{(n-1)} \tag{2.2}$$

where we define

$$p_{ij}^{(0)} = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases}$$

From linear algebra we recognize the relation 2.2 as the formula for matrix multiplication, so that $\mathbf{P}^{(n)} = \mathbf{P} \times \mathbf{P}^{(n-1)}$. By iterating this formula, we obtain

$$\mathbf{P}^{(n)} = \mathbf{P} \times \mathbf{P} \times \ldots \times \mathbf{P}.$$

In other words, the n-step transition probabilities $p_{ij}^{(n)}$ are the entries in the matrix $\mathbf{P}^n$, the nth power of $\mathbf{P}$.

Proof. The proof proceeds via a *first step analysis*, a break down of the possible transitions, followed by an application of the Markov property. The event of going from state i to state j after n transitions can be realised in the mutually exclusive ways of going to some intermediate state $k (k = 0, 1, \ldots)$ after the first transition, and then going from state k to state j in the remaining $(n-1)$ transitions [HP92].

Because of the Markov property, the probability of the second transition is p_{kj}^{n-1} and that of

the first is p_{ik}. If we use the law of total probability, then Equation 2.2 follows [TK14]. The steps are,

$$\begin{aligned} p_{ij}^{(n)} &= P(X_n = j \mid X_0 = i) = \sum_{k=0}^{\infty} P(X_n = j, X_1 = k \mid X_0 = i) \\ &= \sum_{k=0}^{\infty} P(X_1 = k \mid X_0 = i) P(X_n = j \mid X_0 = i, X_1 = k) \\ &= \sum_{k=0}^{\infty} p_{ik} p_{kj}^{(n-1)} \end{aligned}$$

Let m and n be two positive integers and assume $X_0 = i$. In order to get to state j in $(m+n)$ steps, the chain will be at some intermediate state k after m steps. To obtain $p_{ij}^{(m+n)}$, we can obtain the **Chapman-Kolmogorov equation** over all possible intermediate states:

$$p_{ij}^{(m+n)} = P(X_{m+n} = j \mid X_0 = i) = \sum_{k \in S} p_{ik}^{(m)} p_{kj}^{(n)}$$

□

2.5.4 Steady State and Stationary Distribution

Next we consider the long-term behaviour of Markov chains. In particular, we are interested in the proportion of time that the Markov chain spends in each state when n becomes large [PN16]. A Markov chain's equilibrium requires that: (i) the $p_{ij}^{(n)}$ have settled to a limiting value; (ii) this value is independent of the initial state; and (iii) the $\pi_j^{(n)}$ also approach a limiting value π_j [HP92].

Formally, the probability distribution $\boldsymbol{\pi} = [\pi_0, \pi_1, \pi_2, \ldots]$ is called the limiting distribution of the Markov chain X_n. We have,

$$\pi_j = \lim_{n \to \infty} p_{ij}^{(n)},$$

for all $i, j \in S$, with

$$\sum_{j \in S} \pi_j = 1,$$

since,

$$p_{ij}^{(n+1)} = \sum_k p_{ik}^{(n)} p_{kj},$$

as $n \to \infty$,

$$\begin{aligned} \pi_j &= \sum_k \pi_k p_{kj}, \\ \pi &= \pi \mathbf{P}. \end{aligned} \tag{2.3}$$

As we can see the π in Equation (2.3) is the left eigenvector of $\mathbf{P}$ with eigenvalue 1. Therefore, we can solve the limiting distribution by computing the eigenvector of the transition probability matrix $\mathbf{P}$.

2.6 Queueing Theory

In many aspects of daily life, queueing phenomena may be observed when service facilities (counters, web services) cannot serve their users immediately. Queueing systems are mainly characterised by the nature of how customers arrive at the system and by the amount of work that the customers require to be served [Che07]. Another important characteristic is described as the service discipline, which refers to how the resources are allocated to serve the customers. The interaction between these features has a significant impact on the performance of the system and the individual customers.

The first queueing models were conceived in the early 20th century. Back in 1909, the Erlang loss model, one of the most traditional and basic types of queueing models, was originally developed for the performance analysis of circuit-switched telephony systems [Erl09]. Later, queueing theory was successfully applied to many areas, such as production and operations management. Nowadays, queueing theory also plays a prominent part in the design and performance analysis of a wide range of computer systems [Che07].

A queueing system consists of customers arriving at random times at some facility where they receive service of a specific type and then depart. Queueing models are specified as A/S/c,

according to a notation proposed by D. G. Kendall [Ken53]. The first symbol A reflects the arrival pattern of customers. The second symbol S represents the service time needed for a customer, and the 'c' refers to the number of servers. To fully describe a queueing system, we also need to define how the server capacity is assigned to the customers in the system. The most natural discipline is the first come first serve (FCFS), where the customers are served in order of arrivals. Another important type of queueing discipline arising in performance analysis of computer and telecommunication systems is so-called the processor-sharing (PS) discipline, whereby all customers are served in parallel [Che07, HP92]. In this dissertation, we use both disciplines to model systems of interest.

2.6.1 Important Performance Measures

Queueing models assist the design process by predicting and managing system performance. For example, a queueing model might be employed to evaluate the cost and benefit of adding an additional server to a system. The models facilitate us to calculate system performance measures both from the perspective of the system, e.g. overall resource utilisation, or from the perspective of individual customers, e.g. response time. Important operating characteristics of a queueing system include [HP92]:

- λ: the average arrival rate of customers (average number of customers arriving per unit of time)
- μ: the average service rate (average number of customers that can be served per unit of time)
- ρ: the utilisation of the system, indicating the load of a queue
- L: the average number of customers in the system
- L_Q: the average number of customers waiting in line
- P_n: the probability that there are n customers in the system at a given time

2.6.2 Processor-sharing Queueing Models

In the field of performance evaluation of computer and communication systems, the processor-sharing discipline has been extensively adopted as a convenient paradigm for modelling capacity sharing. PS models were initially developed for the analysis of time-sharing in computer communication system in the 1960s. Kleinrock [Kle64, Kle67] introduced the simplest and best-known egalitarian processor-sharing discipline, in which a single server assigns each customer a fraction $1/n$ of the server capacity when $n > 0$ customers are in the system; the total service rate is then fairly shared among all customers present.

An M/M/1 queue is the simplest non-trivial queue where the requests arrive according to a Poisson process with rate λ, which is the inter-arrival times are independent, exponentially distributed random variables with parameter λ. The service time are also assumed to be independent and exponentially distributed with parameter μ. Similar to M/M/1-FIFO, the arrival process of an M/M/1-PS queue is a Poisson process with intensity λ and the distribution of the service demand is also an exponential distribution $Exp(\mu)$. Then the number of customers in the system obeys the same birth-death process as in the M/M/1-FIFO queue [Vir16]. With n customers in the system, the queue length distribution of the PS queue is the same as for the ordinary M/M/1-FIFO queue,

$$\pi_n = (1 - \rho)\rho^n, \quad \rho = \lambda/\mu$$

Accordingly, the expected number of customers in the system, $E[N]$, and by Little's Law[10], the expected delay in the system $E[T]$ are,

$$E[N] = \frac{\rho}{1 - \rho}, \quad E[T] = \frac{1/\mu}{1 - \rho}$$

[10] "Little's law," in *Wikipedia: The Free Encyclopedia*; available from `https://en.wikipedia.org/wiki/Little's_law`; retrieved 12 June 2016.

The average sojourn time of a customer with service demand x is:

$$T(x) = \frac{x}{C(1-\rho)}$$

where C is the capacity of the server. This means that jobs experience a service rate of $C(1-\rho)$ [HP92, Vir16].

2.6.3 BCMP Network

Queueing networks are the systems in which individual queues are connected by a routing network. In the study of queueing networks one typically tries to obtain the equilibrium distribution of the network[11]. However, so as to enable computation of performance metrics such as throughput, utilisation and so on, attempting to solve for the steady state distribution of all queues joint often proves intractable on account of the state space explosion problem. For efficient solution, separable approaches that analyse queues in isolation and then combine the results to obtain a joint distribution are necessary. The leading example of this is the BCMP theorem which defines a class of queueing networks in which the steady state joint probability distributions have a product form solution. It is named after the authors of the paper where the network was first described [BCMP75]: Baskett, Chandy, Muntz and Palacios. The theorem was described in 1990 as "one of the seminal achievements of queueing theory in the last 20 years" by J.M. Harrison and R.J. Williams in [HW90].

Definition of a BCMP network. A network of m interconnected queues is generally identified as a BCMP network if each of the queues is one of the following four types:

1. FCFS with class-independent exponentially distributed service time
2. PS queue
3. IS (Infinite Server) queue

[11] "Queueing theory," in *Wikipedia: The Free Encyclopedia*; available from `https://en.wikipedia.org/wiki/Queueing_theory`; retrieved 12 June 2016.

4. LCFS (Last Come First Serve) with pre-emptive resume

In the final three cases, the service time distribution must have a rational Laplace transform [HP92]. That is,

$$L(s) = \frac{N(s)}{D(s)}$$

where $N(s)$ and $D(s)$ are polynomials in s. Also, it must meet the following conditions:

- External arrivals to node i (if any) follow a Poisson process.
- A customer completing service at queue i will either move to some new queue j with (fixed) probability p_{ij} or leave the system with probability $1 - \sum_{j=1}^{M} p_{ij}$, which is non-zero for some subset of the queues[12].

Also it is assumed that the routing of jobs among queues is state-independent. That is, jobs are routed among the queues according to fixed probabilities (which could be different for different classes of jobs) and not based on the number of jobs in the queues. Then the steady-state joint probability distribution $\pi_{n_1 n_2 \ldots n_M}$ is of the form,

$$\pi_{n_1 n_2 \ldots n_M} = \frac{\pi_1(n_1)\pi_2(n_2)\ldots\pi_M(n_M)}{K}$$

where $\pi_m(n_m)$ is the stationary probability of observing n_m jobs in queue m if the queue were in isolation with a Poisson input process having the same rate as the throughput for queue m, and K is a normalisation constant [HP92]. In the case of an open network, K is always 1. For a closed network, K is determined by the constraint that the state probabilities sum to 1,

$$K = \sum_{\text{all states}} \pi_1(n_1)\pi_2(n_2)\ldots\pi_M(n_M)$$

[12] "BCMP network," in *Wikipedia: The Free Encyclopedia*; available from `https://en.wikipedia.org/wiki/BCMP_network`; retrieved 21 June 2016.

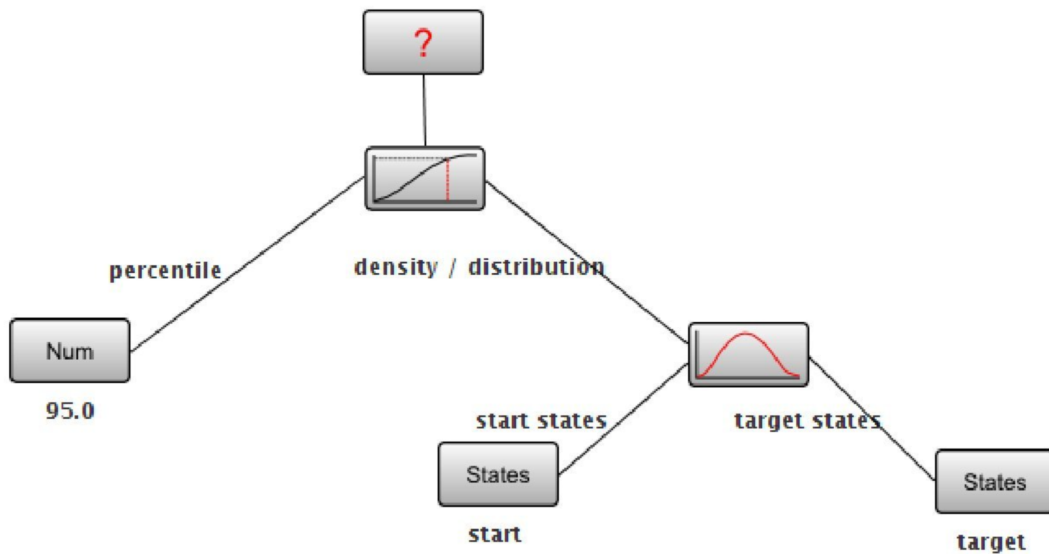


Figure 2.5: An example of Performance Tree query: "what is the 95th percentile of the passage time density of the passage defined by the set of start states identified by label 'start' and the set of target states identified by label 'target'?

2.7 Performance Trees

Finding a way to represent complex performance queries on models of systems in a way that is both accessible and expressive is a major challenge in performance analysis [SBK06]. This section describes an intuitive way to address this challenge via the graphical Performance Trees (PTs) formalism. PTs are designed to be accessible by providing intuitive query specification, expressive by being able to reason about a broad range of concepts, extensible by supporting additional user-defined concepts, and versatile through their applicability to multiple modelling formalisms [KDS09]. Whereas PT queries can also be expressed in textual form [SBK06], the formalism was primarily designed as a graphical specification technique and we focus on this graphical interface to help performance management.

A PT query consists of a set of nodes that are connected via arcs to form a hierarchical tree structure, as shown in Figure 2.5. Nodes in a PT can be of two kinds: operation or value nodes. *Operation nodes* represent performance concepts and behave like a function, taking one or more child nodes as arguments and returning a result. Child nodes can return a value of an appropriate type or value nodes, which can represent other operations such as states, functions on states, actions, numerical values, numerical ranges or Boolean values. PT queries

can be feasibly constructed from basic concepts by linking nodes together into query trees. Arcs connect nodes together and are annotated with labels, which represent roles that child nodes have for their parent nodes. The root node of a PT query represents the overall result of the query. The type of the result is determined by the output of the root's child node while PTs are evaluated from the bottom-up [KDS09, SBK06]. For example, the bottom right of Figure 2.5 computes the probability density function (PDF) of a response time from a set of start states to a set of target states. Then the percentile node extracts the desired percentile i.e. 95th with the response time PDF to yield the 95th percentile of the response time.

PTs have several advantages over current methods for performance query specifications:

Expressiveness: They are represent both performance requirements (that is, Boolean SLO satisfaction) and performance measures in a single query [KDS09]. PTs can represent a broad range of performance-related concepts and operations in their queries, such as availability, reliability and response time/throughput of key services.

Versatility: Different modelling formalisms represent reason about system states and actions differently. PTs feature an abstract state specification mechanism to ensure its versatility. This supports the reasoning about system states in performance queries through state labels. A state label is a user-defined string that identifies sets of system states through a set of associated constraints on the underlying system that are suitable to the modelling formalism applied. [KDS09].

Accessibility: A certain amount of statistical and engineering background is necessary for the understanding and use of PTs; however, such a background is normally characteristic of the target audience of system designers and engineers. PTs further ensure ease of use through their graphical nature allowing for convenient visual composition of performance queries [SBK06].

2.8 Related Research

2.8.1 Cloud Benchmarking Systems

The first part of this dissertation introduces a Performance Tree-based monitoring platform for clouds. While the use of cloud services for either web applications or data analysis has been widely recognised and studied, we have recently seen an explosion in the number of systems developed for benchmarking cloud-based applications and services. Qing et al. [ZCW+13] develop a benchmarking tool, named COSBench, for cloud object storage to help evaluate and compare different object storage systems. The authors show how hardware profiling data can be used to identify performance bottlenecks. Our platform shares the same goal of facilitating system tuning and optimisation via comprehensive hardware and application profiling. By contrast, we focus on providing users an intuitive way of building various performance queries that source from both on-line and historical profiling data, and utilise the performance evaluation result for automatic resource control. YCSB (Yahoo! Cloud Serving Benchmark) [CST+10] is designed with the purpose of facilitating performance comparisons of the new generation of cloud data serving systems, e.g. Bigtable [CDG+08], PNUTS [CRS+08], Cassandra[13], HBase[14], Microsoft Azure[15], and many others. One contribution of this benchmark system is an extensible workload generator, which can load datasets and execute workloads across a variety of data serving systems. Similar to YCSB, we use a series of bash scripts to facilitate the execution of launching different benchmarks, generating workloads, collecting monitoring data etc. LIMBO [KHK+14] introduces a toolkit for emulating highly variable and dynamic workload scenarios. Kashi et al. [VN10] focus on another important dynamic factor in cloud environments – hardware failure. The authors characterise server repair/failure rates in order to understand the hardware reliability for massive cloud computing infrastructure. They find that hard disks are the most replaced component, not just because they are the most numerous components, but as they are one of the least reliable ones. Carsten et al. [BKKL09] discuss some initial ideas of how a new benchmark system should look like that fits better to the characteristics of cloud computing

[13]The Apache Cassandra Project. `http://cassandra.apache.org/`
[14]The Apache Hbase. `https://hbase.apache.org/`
[15]Microsoft Azure. `https://azure.microsoft.com/`

The authors argue that a benchmark for the cloud should have additional ways for measuring complex architectures and systems in which resources are not constantly utilised. We seek in the benchmark that we develop to meet these requirements. All the components of our platform communicate via a publish-subscribe model, which makes it effortless to scale and to further extend. Also, our system calls the corresponding cloud APIs, either CloudStack or Amazon EC2, and the new scaled instance will be registered directly in the registry centre of our platform for future performance monitoring and evaluation, which eliminates the complexity of different application architectures while providing a centralised resource control even for multi-cloud scenarios.

2.8.2 Performance Modelling of Web Applications in Virtualised Environments

The second part of this dissertation is about understanding, modelling and improving the performance of web application in multicore virtualised environments. We are going to cover these three aspects in the following related work.

Multicore system benchmarking. Veal and Foong [VF07] argue that the key to ensuring that performance scales with cores is to ensure that system software and hardware are designed to fully exploit the parallelism that is inherent in independent network flows. The authors perform a comprehensive performance study on commercial web servers and identify that distributing the processor affinity of NICs to different cores can improve the performance of web applications on multicore systems. Harji et al. [HBB12] examine how web application architectures and implementations affect application performance when trying to obtain high throughput on multicore servers. Their experiments reveal that "the implementation and tuning of web servers are perhaps more important than the server architecture". They find that the key factors affecting the performance of this architecture are: memory footprint, usage of blocking or non-blocking system calls, controlling contention for shared resources (i.e. locks), preventing the use of arbitrary processor affinities, and supporting a large number of simultaneous connections. While we share the same idea of improving the performance of a multicore

system, we further incorporate these features into an analytical model to guide the system reconfiguration. Peternier et al. [PBYC13] present a new profiler for characterising the parallelism level within applications. It generates traces based on kernel scheduler events related to thread state changes, profiles the execution of parallel multi-threaded benchmarks on multicore systems and uses the collected profile to predict the wall time execution of the benchmarks for a target number of cores. Hashemian et al. [HKAC13] characterise the performance of dynamic and static network intensive Web applications on a system with two quad-core processors. The authors show that achieving efficient scaling behaviour entails application specific configurations to achieve high utilisation on multiple cores. Also, the authors observe the single CPU bottleneck caused by the default configuration of the NIC affinity. We observe this bottleneck in a virtualised setting in our work and solve it by distributing the NIC interrupts to all available cores to improve the overall CPU utilisation and network throughput.

Virtual machine performance study. Akoush et al. [ASR+10] study the behaviour of live VM migration using the Xen virtualisation platform. The authors show that the Xen migration architecture does not scale up well with high speed links and implement several optimisations to improve migration throughput. They also introduce two migration simulation models based on different migration processes and memory page dirty rate to predict migration time. Pu et al. [PLM+13] present a series of experiments related to measuring the performance of co-located web applications in a virtualised environment. Cherkasova and Gardner [CG05] present a lightweight monitoring system for identifying the CPU overhead in the Xen control domain when processing I/O-intensive workloads. The authors focus on the CPU overhead caused by a particular VM domain, while we focus on the aggregated overhead caused by the hypervisor and all guest VMs. Kousiouris et al. [KCV11] study a wide range of parameters that could affect the performance of an application when running on consolidated virtualised infrastructures, such as workload types, the effect of scheduling, and different deployment scenarios. The authors use a genetically optimised artificial neural network to quantify and predict the performance of the application for a given configuration and enable the cloud providers to optimise the management of the physical resources according to the prediction result. Chiang et al. [CHHW14] present Matrix, a performance prediction and resource management system. Matrix also uses machine

learning techniques and an approximation algorithm to build performance models. It needs a workload classifier to identify new workloads that are running in the guest VMs while our model can adapt to the new workloads by an automatic on-line parametrisation mechanism. Chow et al. [CMF+14] show how to automatically construct a model of request execution from pre-existing logs. This can be achieved by generating a significant number of potential hypotheses about program behaviour and rejecting hypotheses contradicted by the empirical observations. The model for the application behaviour includes the causal relationships between different components and can be constructed for analysing the performance of concurrent distributed systems. This approach points out a new direction to deal with performance analysis in large-scale dynamic environments.

Multicore modelling. Most queueing network models represent k-core processors as M/M/k queues. M/M/k models have also been used when modelling virtualised applications running on multicore architectures. Cerotti et al. [CGPS13] benchmark and model the performance of virtualised applications on a multicore environment using an M/M/k queue. The model is able to obtain good estimates for the mean end-to-end response time for CPU intensive workloads. In our model, we provide a mutli-class queueing network model for abstracting not only CPU/network-intensive workloads but also the imbalanced CPU usage when processing I/O requests. Brosig et al. [BGHK13] predict the overhead of virtualised applications using a multi-server queueing model, which is similar to an M/M/k queue with additional scheduling mechanisms for overlapping resource usage. The authors assume that the relevant model parameters, such as the number of VMs, the VM-specific CPU demands and the VM-specific overhead in terms of induced CPU demand on Dom0 are known, and use extra tools (e.g. ShareGuard in [GCGV06]) to obtain these overhead parameters. They report accurate prediction of server utilisation; however, significant errors occur for response time calculations for multiple guest VMs. Bardhan et al. [BM13] develop an approximate two-level single-class queueing network model to predict the execution time of applications on multicore systems. The model captures the memory contention caused by multiple cores and incorporates it into an application-level model. The authors also use hardware counters provided by the Intel processor to parameterise the memory contention model; however, such counters are usually difficult to obtain

when the servers are virtualised on a hypervisor. Deng et al. [DP11] tackle the performance optimisation problem using queueing analysis. The multicore pipline processes are analysed in a tandem queueing model[16]. They conduct simulations to review the best models derived from the optimal average time in the model.

2.8.3 Performance Interference Modelling in Multi-Tenant Clouds

The third part of this dissertation is about diagnosing and managing performance interference in multi-tenant Clouds. We discuss the related work in the following.

Hypervisor overhead benchmarking is an essential part of performance study and analysis. Cherkasova et al. [CG05] present a lightweight monitoring framework for measuring the CPU usage of different VMs including the CPU overhead in the control domain caused by I/O processing. The performance study attempts to quantify and analyse this overhead for I/O-intensive workloads. Also, the authors analyse the impact of different Xen schedulers on application performance and discuss challenges in estimating application resource requirements in virtualised environments. Shea et al. [SWWL14] study the performance degradation and variation for TCP and UDP traffic, then provide a hypervisor reconfiguration solution to enhance the overall performance. Pu et al. [PLM+13] present experimental research on performance interference on CPU and network intensive workloads on the Xen hypervisor and reach the conclusion that identifying the impact of exchanged memory pages is essential to the in-depth understanding of interference costs in Dom0. We also find this is critical to maintain the performance of I/O-intensive workloads in our benchmark experiments. Barker et al. [BS10] conduct a series of empirical studies to evaluate the efficacy of cloud platforms for running latency-sensitive multimedia applications. The study focuses on whether dynamically varying background load from such applications can interfere with the performance of latency-sensitive tasks. The EC2 experiments reveal that the CPU and disk jitter and the throughput seen by a latency-sensitive application can indeed degrade due to background load from other VMs. Their hypervisor experiments also indicate similar fluctuation when sharing the CPU and the authors

[16] A finite chain or a loop of queues where each customer visit each queue in order

observe fair throughput when CPU allocation is pinned by the hypervisor. Their experiments also reveal significant disk interference, resulting in up to 75% degradation under sustained background load. We have the same configuration for conducting our benchmark experiments, and our work makes similar findings but then uses those to guide our analytical model. Due to the combination effect of hypervisor and other VMs, acquiring reliable performance measurements of VMs becomes harder than on a physical server. Shadow Kernels [CCS+15] is proposed to avoid this problem by forgoing hypervisor fidelity and using the hypervisor to provide a performance measurement technique that cannot be used on bare metal.

Performance interference modelling. Most work on performance interference modelling is based on machine learning, heuristic methods, and queueing models. Nathuji et al. [NKG10] use online feedback to build a multiple-input multiple-output model that captures performance interference interaction. The implementation and the evaluation of this work only focuses on CPU-bound applications while we consider different types of workloads. Caglar et al. [CSG13] propose a machine learning-based technique to classify the performance of VMs based on historical mean CPU and memory usage, and extract the patterns that provide the lowest performance interference while still allowing resource overbooking. The dataset for classification and pattern training sources from the Google cluster trace log, and an artificial neural network model is proposed to capture the performance interference among VMs in terms of CPU and memory capacity. Kang et al. [KKEY12] propose a consolidation mechanism by exploring the performance impact of contention in the last-level shared cache (LLC). In our experiment, we try to reduce this impact by pinning virtual CPUs (vCPUs) to physical CPUs to let vCPUs run on specific NUMA nodes. Casale et al. [CKK11] propose a simple linear prediction model to predict the throughput and response time degradation of mixed read/write disk requests due to storage device contention in consolidated virtualised environments. Lim et al. [LHK+12] introduce a dilation-factor based model to derive the completion time of jobs in shared service systems. This is similar to the idea of our virtualisation slowdown factor in Chapter 5. The main differences between their work and our work is that our model not only considers the contention of systems with multiple resources, but also takes the hypervisor scheduling effects into account. Govindan et al. [GLKS11] propose a technique for predicting performance interfer-

ence due to a shared processor cache. The technique includes using a synthetic cache loader to profile an application's cache usage, creating an application clone, building degradation tables and predicting a candidate placement by looking up the performance degradation tables. By contrast, we aim to propose a lightweight method to identify, predict and minimise performance interference without comprehensive micro-benchmarks or online training to keep the approach simple and feasible.

Scheduling and consolidation. To avoid excessive interference between VMs, scheduling and consolidation decisions are critical. Urgaonkar et al. [UWH+15] propose a dynamic service migration and workload scheduling control policy to optimise the network operational cost while providing rigorous performance guarantees based on the analysis of a Markov Decision Process (MDP). The authors develop a series of novel techniques solving a class of constrained MDPs that possess a decoupling property, and design an online control algorithm to achieve cost-optimal placement. The method does not require any statistical knowledge of the system parameters. Roytman et al. [RKG+13] propose a polynomial time algorithm to yield a solution determining the best suited VM consolidation within performance constraints. However, they propose is an offline-based approach, while we seek to provide on-line performance degradation prediction and resource consolidation in an online manner. Chiang et al. [CH11] present the TRACON system, a task and resource allocation control framework that mitigates the interference effects from concurrent data-intensive applications and improves the application performance. Kim et al. [KEY13] present a VM consolidation method based on the idea that a highly interference-sensitive VM should be co-located with less interference-sensitive ones. Ajay et al. [GSA+11] present Pesto, an automated and on-line storage management system for virtualised data centres. Similar to our work which uses the model results to derive scheduling mechanism, their experimental evaluation on a diverse set of storage devices demonstrates that Pesto's online device model is accurate enough to guide storage planning and management decisions. By contrast, our VM placement decision is proposed for consolidating any type of applications in Clouds. Farley et al. [FJV+12] show how cloud customers can deliberately guide the placement of their workloads to improve performance by applying gaming strategies. To eliminate performance interference in shared data centres, Angel et al. [ABK+14] present Pul-

sar, a system that enables data centre operators to provide guaranteed end-to-end performance isolation. Pulsar proposes virtual data centres (VDCs) as an abstraction which encapsulates the performance guarantees given to tenants. The design of Pulsar consists of a centralised controller with full visibility of the data centre topology and tenants' VDC specifications and a rate enforcer inside the hypervisor at each compute server. We share a similar design for identifying the guests' performance. In our CloudScope system, we use a virtualisation slowdown factor to indicate performance interference. However, instead of placing a rate enforcer at the hypervisor, our Dom0 controller automatically configures the hypervisor on-the-fly to provide performance guarantees to the guest VMs.

Chapter 3

A Performance Tree-based Monitoring Platform for Clouds

3.1 Introduction

This chapter presents a performance tree-based monitoring and resource control framework. It begins with an overview of the system design requirements then discusses the system architecture and different components. We next present some lessons we learned for reproducible and reliable application measurements in virtualised environments. Finally, we show how our platform captures the SLO violations of applications running in clouds and automatically scales out the virtual resource.

Cloud-based software systems are expected to deliver reliable performance under dynamic workload while efficiently managing resources. Conventional monitoring frameworks provide limited support for flexible and intuitive performance queries. We develop a prototype monitoring and control platform for clouds that is a better fit to the characteristics of cloud computing (e.g. extensible, user-defined, scalable). Service Level Objectives (SLOs) are expressed graphically as Performance Trees, while violated SLOs trigger mitigating control actions.

Active performance management is necessary to meet the challenge of maintaining QoS in cloud

environments. In this context, we present a Performance tree-based monitoring and automatic resource control framework for clouds which makes several contributions:

- We outline system requirements for an extensible modular system which allows for monitoring, performance evaluation and automatic scaling up/down control of cloud-based Java applications.
- We introduce a series of methodologies to produce reliable measurements.
- We present a front-end which allows for the graphical specification of SLOs using PTs. SLOs may be specified by both live and historical data, and may be sourced from multiple applications running on multiple clouds.
- We demonstrate how our monitoring and evaluation feedback loop system ensures the SLOs of a web application are achieved by auto-scaling.

3.2 The Design of a Performance Tree-based Monitoring Platform

3.2.1 System Requirements

In this section, we present an overview of our system by discussing the requirements for an effective cloud-based monitoring platform and the techniques used to achieve them.

In-depth performance profiling. We require the ability to extract generic metrics on a per-application basis, such as CPU utilisation, memory usage, etc., as well as custom application-specific metrics [ZCW+13]. This functionality is best delivered through a well-defined API.

Accessible performance query specification & online/offline evaluation. An important feature which distinguishes our platform from other available tools is the fact we use Performance Trees (PTs) [SBK06, DKS09] for the graphical intuitive, and flexible definition of

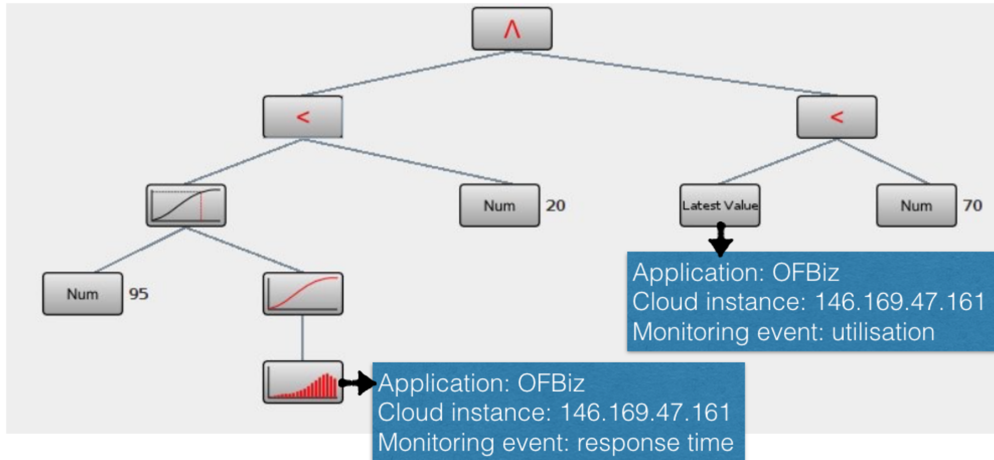


Figure 3.1: An example of Performance Tree-based SLO evaluation : "With respect to Apache OFBizs 146.169.47.121 instance, is it true that 95% of observed response times are smaller than 20ms and the utilisation of the machine is smaller than 70%"?

performance queries and evaluation, while most of the readily available monitoring tools provide users with a textual query language [GBK14]. We also incorporate support for historical trend analysis by retaining past performance data. A performance query which is expressed in a textual form as follow can be described as a form of hierarchical tree structure, as shown in Figure 3.1. Figure 3.1 shows a fully constructed PT query in GUI, which demonstrates an on-line SLO evaluation on both response time and utilisation of the corresponding server.

Extensible & Scalable. First, since multiple applications and multi-cloud environments may impose different choices of programming language and monitoring tool, a light-weighted platform independent data format (i.e. JSON) is used for monitoring data exchange. Second, all components of our system communicate using a publish-subscribe model, which allows for easy scaling and extensibility. Third, a NoSQL database is used due to ability to support large data volumes found in real-world use cases [BBM+13].

3.2.2 System Architecture

Figure 3.2 illustrates the architecture of our system. To provide users with a complete framework that is able to monitor application performance, design performance performance models, specify performance queries in the form of PTs, evaluate these and provide resource scaling

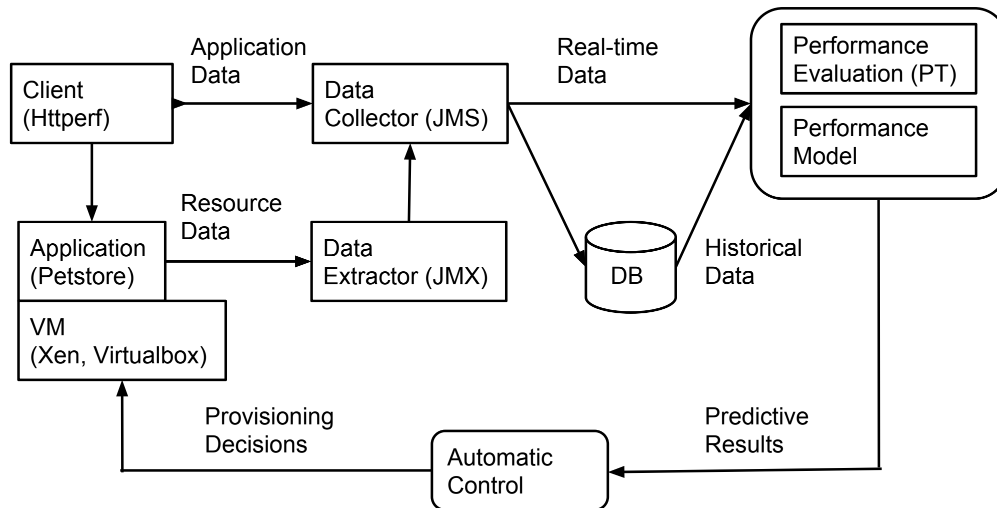


Figure 3.2: System architecture

based on the evaluation result, we have developed a performance analysis platform that supports this functionality in an integrated manner. The system is realised as a Java application capable of monitoring and evaluating the performance of a target cloud application, with concurrent scaling of the cloud system so as to meet user-specified SLOs.

All components of our platform communicate using a publish-subscribe fashion. The use of event streams can help make the system more scalable, more reliable, and more maintainable. People now are excited about these ideas because they point to a future of simpler code, better robustness, low latency and more flexible for doing interesting things with data [Kle16]. The *data collector* (Java Message Service) extracts application metrics, e.g. response time, throughput and job completion time. This is combined with the output of the *data extractor* (Java Management Extension), which provides hardware-related metrics, i.e. utilisation of each core of the VM, memory bandwidth, etc. The data collector is responsible for aggregating all the performance data published by applications running on monitored VMs. It can either feed this data directly into the *performance evaluator* or store it a database for future analysis. The performance evaluator evaluates metrics starting from the leaves of the PTs and ending with the root, thus producing performance indices which are compared to target measurements for resource management. The *automatic controller* (autoscale) then optimises the resource configuration to meet the performance targets [CHO+14].

3.3 The Myth of Monitoring in Clouds

Performance analysis and resource management begin with the monitoring of system behaviour. There is a rich history of methods and techniques that understand, profile and troubleshoot system performance, both in practice and in the research literature as mentioned in Section 2.8. Yet, most of these prior techniques do not deal sufficiently with the complexities that arise from virtualised cloud infrastructures. The complexity comes from the additional virtualisation layer and performance delay can be caused by different components, e.g. virtual blocks, hypervisor, or the communication between these components. In this section, we introduce a series of performance analysis methodologies for measuring and uncovering performance insights into complex virtualised systems. It is also important to understand the sensitivity of the system by tuning different parameters. This gives a good understanding of the workload when comparing different results or creating experiments that are easily reproducible.

3.3.1 Measuring without Virtualisation

We start with measuring I/O performance on a bare metal as an example. Modern storage systems are a complex mixture of potentially heterogeneous storage devices, networking hardware and layers of cache. Their performance is driven by the underlying service capacity and the characteristic of the workload. Examples of important workload characteristics are:

- Workload intensity – Changes in workload intensity, such as in the number of active users and arrival rates of concurrent requests, can significantly influence the behaviour of systems. Response times can grow exponentially with increasing workload intensity. The workload intensity of many systems also show periodic patterns over the day and week [Roh15].

- Access pattern – The total throughput that an application can achieve depends on the access pattern the application generates on the disk subsystem. For example, a mail server might have 67% random reads and 33% random writes with an 8 KB block size;

while a video streaming server might have 100% sequential reads with large block size greater than 64 KB [HBvR+13].

Examples of factors which influence servers capacity are:

- Block sizes – A data transfer is performed in blocks when a disk is accessed. The size of the transferred data blocks depends on the features of the operating system and the application [WID+14]. To have a better understanding of how the disks behave, we issue requests with different block sizes, varying from 512 bytes to 4 MB in our benchmarking experiments. This provides a good representation of the performance delivered by the storage infrastructure.

- Different LBAs (logical block addressing) – In the ZCAV (Zoned Constant Angular Velocity) scheme, a disk is divided into a series of regions. As a result, disks perform differently in different regions. It is necessary to read and write to numerous different zones in order to obtain a reliable model of performance for the disk [LDH+09].

- IQ queue depth – To understand the effect of I/O queue depth, we show a screen-shot of iostat (see Figure 3.3) when running the fio benchmark (a flexible I/O benchmark[1]) in an 8-core PM as an example. Fio is capable of generating I/O workloads based on parameters describing, such as read/write mix, queue depth, request size, and sequentially/randomly [WID+14]. The screen-shot displays the information about CPU usage and I/O statistics about all the devices in the system. We explain the detail of each column as follows:

  - "`avgqu-sz`"– average queue size is the average I/O waiting time in the queue.
  - "`r_await`" and "`w_await`"– host read and write time (in milliseconds) are the times that read and write I/Os spend in the I/O queue.
  - "`svctm`"– service time (in milliseconds) is the time it takes to send the I/O request to the storage and get a response back – that is, the time the storage subsystem requires to handle the I/O.

[1] fio. `http://freecode.com/projects/fio`

```
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          14.37    0.00    1.64   18.90    0.00   65.09

Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0.00    13.50 1980.50  325.50  1980.50  1573.25     3.08    83.25   36.10    4.02  231.30   0.43 100.00
dm-0              0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00    0.00    0.00    0.00   0.00   0.00
dm-1              0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00    0.00    0.00    0.00   0.00   0.00
dm-2              0.00     0.00    0.00  335.00     0.00  1562.00     9.33    78.62  234.68    0.00  234.68   1.72  57.60
dm-3              0.00     0.00 1980.50    3.50  1980.50    11.25     2.01     7.99    4.03    4.02    7.43   0.50 100.00
```

Queueing
Host wait
Storage wait
Utilisation
Reads
Writes

Figure 3.3: The screenshot of iostat running fio benchmark tests

We consider a simple example to explain the relationship between these three metrics: Assume that a storage system manages each I/O in exactly 10 milliseconds and there are always 10 I/O requests in the queue; then the average wait time for a newly arriving I/O request will be $10 \times 10 = 100$ *milliseconds*. In that case, we can verify by Little's Law that `storage wait` * `queue size` $\approx$ `host wait`. So the actual delay of an I/O request is the reported average wait time plus the reported service time. Some measurement systems will report either of the higher, or the lower numbers with respect to the actual response time. It is necessary to make sure which statistics are used when analysing I/O performance [CGPS13].

- Local cache control. Some benchmarking tools communicate with storage directly while others go through the local file system. Additionally, they can be configured to cache data or not using the `O_DIRECT` flag [RNMV14]. It is important to understand the exact way the tool is performing I/O.

3.3.2 Measuring with Virtualisation

While the nature of data centre operations has changed, the measurement and management methodology for VMs has not been adapted adequately [GSA+11]. We found that some of the existing techniques do not work properly in virtualised environments. This makes performance management difficult as the input data is critical to make correct decisions, such as resource

allocation and configuration. A recent Google analysis [KMHK12] presents a prime example of the effect of this limitation, whereby a significant portion of Google's production systems became unpredictable and unresponsive due to misconfiguration of the underlying systems. We present some of the lessons we learned and how we collect the data in the data extractor component (see Figure 3.2) to conduct reliable and reproducible monitoring in virtualised environments. The main areas to pay attention to are:

1. Hypervisor:

- Give Dom0 enough vCPUs – When the PM hosts network or storage I/O-intensive workloads, a large portion of the processing will happen in Dom0. It is important that Dom0 has enough vCPUs and these cores are configured to deliver the power the guest VMs require from them [XBNJ13]. The actual number will depend on your environment: the size of NUMA nodes, the amount of other guests that plan to run, etc.
- Control C/P states and turbo mode – Processor performance states (P-states) and processor operating states (C-states) are the capability of a processor to switch between different supported operating frequencies and voltages to modulate power consumption[2]. For example, a processor in P3 state will run more slowly and use less power than a processor running at P1 state, and at higher C-states, more components shut down to save power [GNS11b]. For rigorous measurements, it is recommended to turn hyper-threading, C/P states and turbo off to avoid system from behaving differently under different modes. With hyper-threading turned off, stress on the first level cache lines is also reduced.
- Change all VMs to PV mode to avoid QEMU processes – Xen can also run in HVM mode as we discussed. Hardware is emulated via a QEMU device model daemon running as a backend in Dom0[3] It is recommend to disable the QEMU process (i.e. `qemu-dm`) while conducting experiments in a PV mode VM.
- Give Dom0 dedicated memory – It is important to make sure that Dom0 memory is

[2]CPU performance states (P-states) and CPU operating states (C-states). `https://www.ibm.com/support/knowledgecenter/linuxonibm/liaai.cpufreq/CPUPerformanceStates.htm`

[3]"QEMU," in *Wikipedia: The Free Encyclopedia*; available from `https://en.wikipedia.org/wiki/QEMU`; retrieved 9 June 2016.

never ballooned down while starting new guests. Dedicating fixed amount of memory for Dom0 is favourable for the two reasons: (i) Dom0 calculates various network related parameters based on the boot time amount of memory [BDF+03] and; (ii) Dom0 needs memory to store metadata and this allocation is also based on the boot time amount of memory [BDF+03, MST+05]. Ballooning down Dom0 might affect the calculation, with bad side effects[4].

2. Individual VMs:

- Fine-tune and pin vCPUs to pCPUs – This is helpful to make sure: (i) interrupts are delivered to the exact cores that the VMs are running [GDM+13] and (ii) the vCPUs running on NUMA hardware can allocate memory closer to its node [NLY+11]. If vCPUs are dynamically swapped between different pCPUs, cache misses are likely to affect performance detrimentally.
- Give the application VMs enough RAM. The whole performance might degrade because the guest VM runs out of memory and starts to swap to disk. It is also recommended to not to give more RAM than the hypervisor is able to physically allocate [RNMV14].

3. Where and how to measure:

- CPU measurements – Xentop is widely used to monitor and record the physical CPU utilisation of each domain, because a top command in each domain can only get the domain's virtual CPU utilisation [CH11, BRX13, GDM+13]. However, within each guest (including the control domain) it is also recommended to run 'top' to keep an eye on certain fields. For example, when running 'top' within a VM, if it reports any number different than zero on the "st" field (steal time), this means that another VM (or even the hypervisor) is "stealing" time from that particular domain. In that case, it is necessary to make sure the VM is running without other VMs or hypervisor stealing its own time by pinning of VMs to PMs if necessary.

[4] Xen Project Best Practices. `http://wiki.xenproject.org/wiki/Xen_Project_Best_Practices`

- I/O measurements in bare metal, Dom0 and guests – To produce reliable I/O measurements in virtualised environments it is useful to run benchmarks on: (i) the bare metal hardware (a typical Linux installation with Xen); (ii) Dom0 and (iii) the ordinary guest domains. Many people believe that running benchmarks from Dom0 is the same thing as running benchmarks from bare metal [MLPS10, SIB+14]. This is only partially correct. In Dom0, every hardware interrupt is, in reality, an event channel notification delivered by the hypervisor (Xen). This includes both timers and interrupts from devices such as hard drives completing requests or NICs handling packets. Even though these processes are fast and low-impact in Xen, there is still some overhead that is noticeable. With the new kernel (i.e. XenServer 6.5 – Creedence), it is feasible to boot a bare-metal kernel without the hypervisor for conducting bare metal experiment comparisons to make sure the I/O requests perform correctly and the performance reach the hardware limit instead of having bottlenecks in the Dom0 or other components.

4. Network-attached storage:

- The iSCSI[5]-attached storage array – If one is present, the target configuration that is negotiated between the host and the storage array should be checked [CWM+14, LDDT12]. Depending on certain parameters, the array might respond differently to different block sizes or concurrent processes.

- It is required to write into the whole virtual disk first from VMs before the actual experiments, because some of the format, e.g. VHD, are very thin-provisioned. When requests first arrive at Dom0, tapdisk[6] (every virtual disk is backed by a tapdisk process) will reply with an array of zeros, because it knows there is no actual data allocated on certain block addresses. When measuring writes, it is also important to measure the time of the first write and the time for subsequent writes separately. Again, due to thin-provisioning, writing the first time takes longer than later as VHD metadata needs to be updated as the virtual disks grow.

[5] iSCSI, Internet Small Computer Systems Interface. `https://en.wikipedia.org/wiki/ISCSI`
[6] Blktap. `http://wiki.xenproject.org/wiki/Blktap`

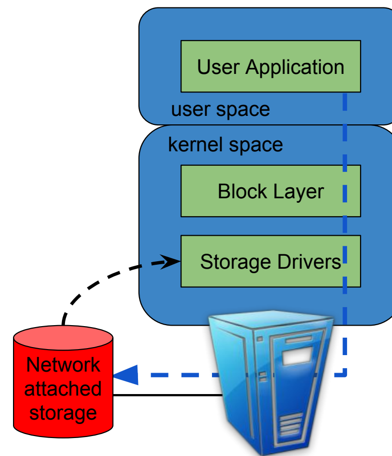


Figure 3.4: The path of an application issuing requests to disks without virtualisation

3.3.3 An Example of Measuring the Actual I/O Queue Size

There are many discussions on technical blogs about 'why I/O queue depth matters' for troubleshooting and measuring storage performance. To understand how the average I/O queue size can be measured correctly, we first consider a Linux system processing I/O requests without virtualisation. Figure 3.4 presents fio that issues requests to a SCSI disk. As shown in the sequence diagram of this process in Figure 3.5 , each request reaches the block layer first and then the device driver. The number of read and write "ticks", the amount of time per request that the device has been occupied to keep the disk busy, are available in the block layers[7]. The block layer starts this accounting immediately before shipping the request to the driver and stops it immediately after the request completed. It contains vertical arrows between the layers representing requests departing from and arriving at different layers. Figure 3.5 represents this time in the red and blue horizontal bars. Bars might overlap with each other if more than one request has been submitted concurrently. The ticks might increase at a greater rate when requests overlap.

As shown in Figure 3.6, I/O requests start in a virtual machine's user space application. When moving through the kernel, however, they are directed to PV storage drivers (e.g. `blkfront`) instead of an actual SCSI driver. These requests are picked up by the storage backend (`tapdisk3`)

[7] https://www.kernel.org/doc/Documentation/block/stat.txt

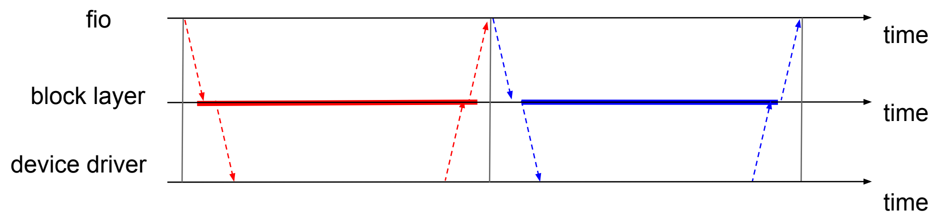


Figure 3.5: The sequence diagram of an application issuing requests to disks

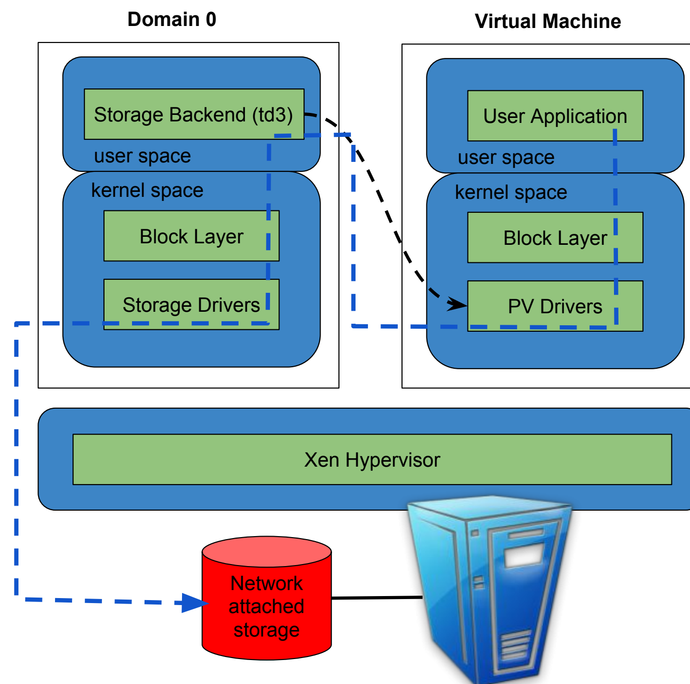


Figure 3.6: The path of an application issuing requests to disks with virtualisation, compiled using information sourced from Citrix virtualisation blog[8]

in Dom0's user space. They are submitted to Dom0's kernel via libaio, pass the block layer and reach the disk drivers for the corresponding storage infrastructure.

Figure 3.7 shows the sequence diagram of processing an I/O request in this case. The technique described in the following to calculate the average queue size will produce different values depending on where in the stack it is applied. For example, when issuing the fio benchmark (e.g. random reading 4 KB requests using `libaio` and with io_depth set to 1), we run iostat

[8] "Average queue size and storage I/O metrics," in *XenServer: Open Source Virtualisation*; available from `http://xenserver.org/discuss-virtualization/virtualization-blog/entry/avgqusz.html`; retrieved 7 June 2016.

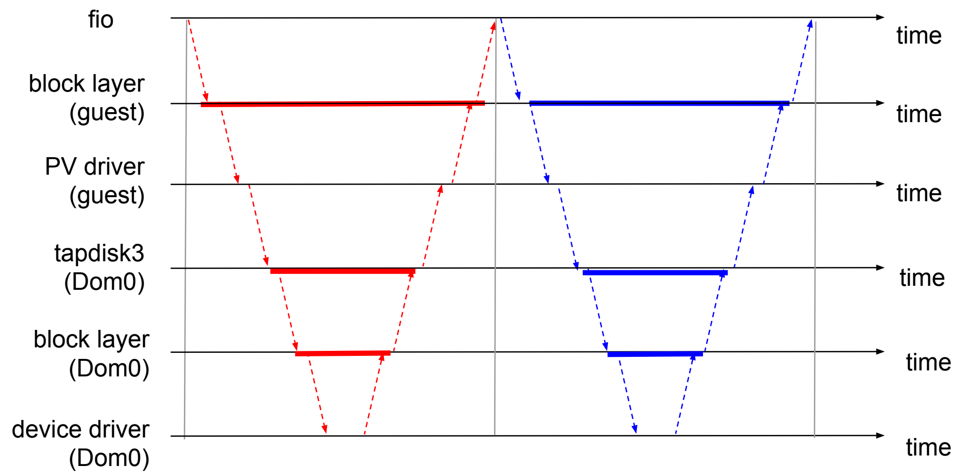


Figure 3.7: The sequence diagram of an application issuing requests to disks with virtualisation, compiled using information sourced from Citrix virtualisation blog[9]

to profile the disk device workload. We show how to obtain the targeted performance metrics within the sequence in Figure 3.7 as follows:

- Within the guest, the value of interest reported in the column "`avgqu-sz`" indicates the queue size of the guest's block layer (guest) as shown in Figure 3.7.
- The next layer of the storage subsystem that accounts for utilisation is `tapdisk3`. The average queue size can be obtained by running `/opt/xensource/debug/xsiostat` in Dom0. This gives an idea of the time that passed between a request being received in the guest's block layer (guest) and in Dom0's backend system (`tapdisk3`).
- Further, it is possible to run iostat in Dom0 and find out what is the perceived queue size at the last layer before the request is issued to the device driver as shown in Figure 3.7 from block layer (Dom0) to device driver (Dom0).

Next, We apply these monitoring techniques to our Performance Tree-based monitoring platform to obtain reliable performance measurements. We will show how our monitoring and evaluation system guarantees the SLOs of a cloud-based application by auto-scaling of resource.

[9] "Citrix virtualisation blog," in *XenServer: Open Source Virtualisation*; available from `http://xenserver.org/discuss-virtualization/virtualization-blog/blogger/listings/franciozzy.html`; retrieved 7 June 2016.

3.4 GUI and Demo in Action

Oracle Java Petstore[10], a typical HTTP-based web application, is used to expose a server to high HTTP request volumes which cause intensive CPU activity related to the processing of input and output packets. The hypervisor (XenServer 6.2) is running on an a Dell PowerEdge C6220 compute server with two Intel Xeon E5-2690 8-core 2.9 GHz processors and two 1 TB hard drives. The network between each server is 10 Gbps. Each server virtual machine is assigned with on vCPU with 1 to 4 cores, 4 GB memory and one vNIC. Httperf is configured on the other servers to send a fixed number of HTTP requests rate incrementally for each Petstore instance [CHO+14].

The user interface contains a dashboard where the user can manage up to 8 different PTs as shown in Figure 3.8. The graphical nature allows easy comprehension and manipulation by the users, and – thanks to their extensibility – new nodes can be added. The user can design their own PTs, either from scratch or by loading in a saved tree from a file. The user can specify the performance metrics, the application, and the server they want to evaluate. Once the tree has been designed and the evaluation has been started, the editor receives data from the data collector or the database, which it uses to update the square panel in the GUI. If the performance requirement is violated, this is represented by a red colour applied to the evaluation box; otherwise, it is green.

Figure 3.8 illustrates two cases: (a) the monitoring and control of a response-time-related SLO with autoscaling enabled, (b) the monitoring of a memory-consumption-related SLO with autoscaling disabled. In the first evaluation, once the PT detects the SLO is violated, the automatic controller module migrates the server to a larger instance. In this case, the server is migrated from a 1 core to a 2 core instance, so the response time decreases and the SLOs is not violated, represented as a 'green' PT block. The migration time is usually around 8 to 10 secs inside of the same physical machine. In the second case, the 'red' block illustrates the memory-consumption-related SLO is violated since autoscaling is not enabled.

[10] Java Petstore. `http://www.oracle.com/technetwork/java/index-136650.html`

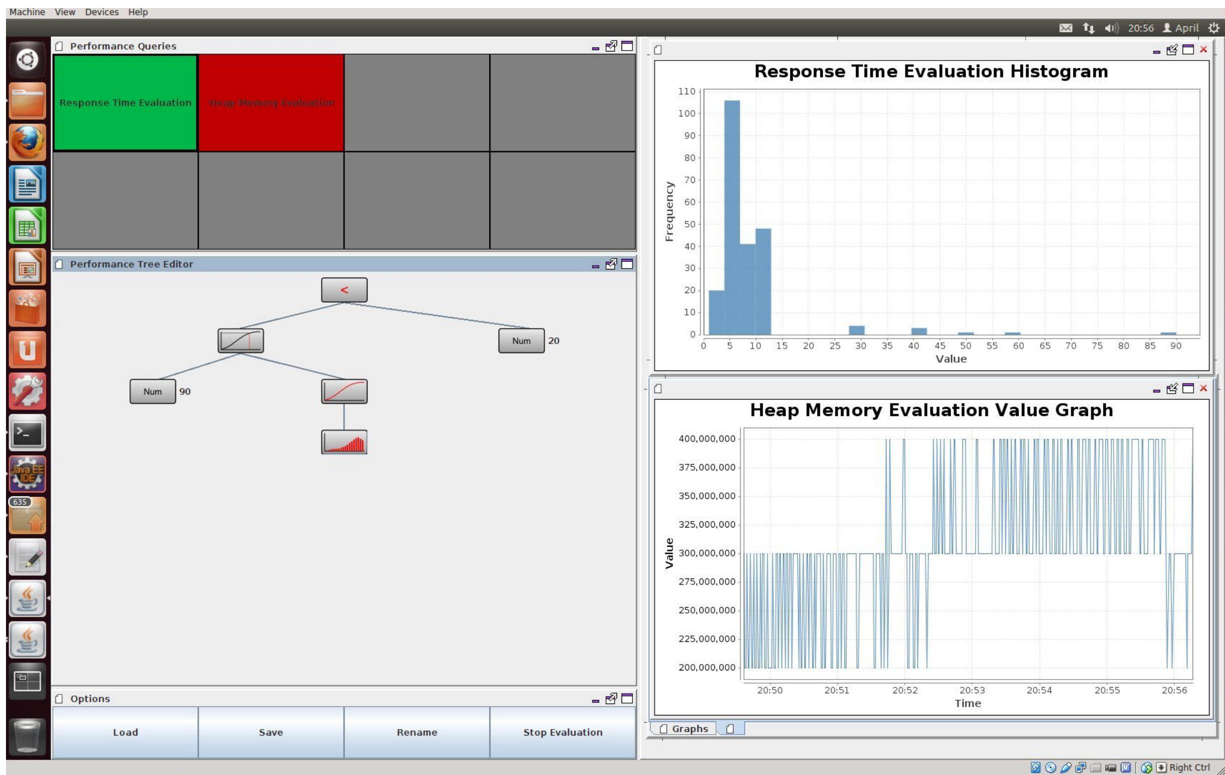


Figure 3.8: A Performance Tree evaluation in progress. Once users create a performance tree and clicks 'Start Evaluation', users receive instant feedback of performance criteria meets or not showing in green or red. Violated SLOs trigger mitigating resource control actions.

3.5 Summary

The current trend of moving business applications into the cloud has exposed the need of monitoring and resource control systems to allow administrators better understand and troubleshoot their applications. We have devised a modular architecture with feedback loop which allows for comprehensive monitoring of cloud-based applications and is readily available for expansion. Another important feature we were able to incorporate into the architecture is the use of Performance Trees as means for defining arbitrary complex performance queries. In this way, we combine a thorough monitoring methodology for virtualised cloud environments. Lastly, we demonstrate that the platform can be successfully used to evaluate SLOs on-the-fly and react to the SLO violations via automatic scaling out the virtual resource.

Chapter 4

Predicting the Performance of Applications in Multicore Virtualised Environments

4.1 Introduction

This chapter presents a modelling technique for predicting the scalability of CPU/network intensive applications running in multicore virtualised environments. Key to our approach is providing a fine-grained model which incorporates the idiosyncrasies of the operating system and the imbalance between multiple CPU cores.

Applications running in cloud environments exhibit a high degree of diversity; hence, strategies for allocating resources to different applications and for virtual resource consolidation increasingly depend on understanding the relationship between the required performance of applications and system resources [SSM+11]. To increase resource efficiency and lower operating costs, cloud providers resort to consolidating resources, i.e. packing multiple applications into one physical machine [CSA+14]. Understanding the performance of these applications is important for cloud providers to maximise resource utilisation and augment system throughput

while maintaining individual application performance targets. Performance is also important to end users, because they are keen to know their applications are provisioned with sufficient resources to cope with varying workloads. Instead of increasing or decreasing the same instances one by one [IDC09], a combination of multiple instances might be more efficient to deal with the burstiness of dynamic workloads [WSL12, SST12, NBKR13]. To handle the resource scaling problems, a model that can appropriately express, analyse, and predict the performance of applications running on multicore VM instances is necessary.

There are at least three observations we can make in light of present research. First, not all workloads/systems benefit from multicore CPUs [FAK+12, TCGK12] as they do not scale linearly with increasing hardware. Applications might achieve different efficiency based on their concurrency level, intensity of resource demands, and performance level objectives [EBA+11]. Second, the effects of sharing resources on system performance are inevitable but not well-understood. The increased overhead and dynamics caused by the complex interactions between the applications, workloads and virtualisation layer introduce new challenges in system management [HvQHK11]. Third, modelling of low-level resources, such as CPU cores, are not generally captured by models [GNS11a, KTD12] or models are not comprehensive enough to support dynamic resource allocation and consolidation [RBG12, TT13].

Many benchmarking studies suggest that each individual core performs differently across the cores of one multiprocessor [JJ09, PJD04, HKAC13]. Veal et al. [VF07] and Hashemian et al. [HKAC13] observe a CPU single core bottleneck and suggest methods to distribute the bottleneck to achieve better performance. However, most modelling work treats each core of a multicore processor equally by using M/M/k queues [CGPS13, BGHK13], where k represents the number of cores. To the best of our knowledge, the problem of modelling the imbalance between cores and the performance of applications in multicore virtualised environment has not been adequately addressed.

This chapter presents a simple performance model that captures the virtual software interrupt interference in network-intensive web applications on multicore virtualised platforms. We first conduct some benchmark experiments of a web application running across multiple cores, and

then introduce a multi-class queueing model with closed form solution to characterise aspects of the observed performance. Target metrics include utilisation, average response time and throughput for a series of workloads. The key idea behind the model is to characterise the imbalance of the utilisation across all available cores, model the processing of software interrupts, and correctly identify the system bottleneck. We validate the model against direct measurements of response time, throughput and utilisation based on a real system. We take steps to alleviate the bottleneck, which turns out to involve at a practical level the deployment of multiple virtual NICs. Analysis of the model suggests a straightforward way to mitigate the observed bottleneck, which can be practically realised by the deployment of multiple virtual NICs within our VM. Next we make blind predictions to forecast performance with multiple virtual NICs. Thereafter, we make blind prediction to forecast the performance of the system with multiple virtual NICs for improved performance.

The rest of the chapter is organised as follows. Section 4.2 presents our testbed setup and performance benchmarking results. Section 4.3 introduces our performance model and validates it. Section 4.4 extends our model for new hardware configurations and Section 4.5 concludes.

4.2 Benchmarking

In this section, we conduct an initial benchmarking experiment to study the impact of multiple cores and software interrupt processing on a common HTTP-based web application. Requests do not involve database access and hence, no disk I/O is required during a response. Our application is the Oracle Java Petstore 2.0[1] which uses GlassFish[2] as the HTTP server. We run the Petstore application on VirtualBox and Xen hypervisor, respectively. The Oracle Java Petstore 2.0 workload is used to expose the VM to high HTTP request volumes which cause intensive CPU activity related to processing of input and output network packets as well as HTTP requests. Autobench[3] was deployed to generate the HTTP client workload.

[1] Java Petstore. `http://www.oracle.com/technetwork/java/index-136650.html`
[2] GlassFish. `https://glassfish.java.net/`
[3] Autobench. `http://www.xenoclast.org/autobench/`

Testbed Infrastructure. We set up two virtualised platforms Xen and Virtualbox as shown in Figure 4.1, using the default configurations. The hypervisors are running on an IBM System X 3750 M4 with four Intel Xeon E5-4650 eight-core processors at 2.70GHz to support multicore VM instances comprising 1 to 8 cores. The server has a dual-port 10 Gbps Ethernet physical network interface card (pNIC), which can operate as a virtual 1 Gbps Ethernet NIC (vNIC). The physical NIC interrupt handling is distributed across the cores, providing maximum interrupt handling performance. The machine is equipped with 48 GB memory and connected to sockets with DDR3-1333MHz channels. The key aspect of our testbed infrastructure that physical CPU cores and network bandwitch are over-provisioned in the physical hardware compared to the corresponding resources in the VMs.

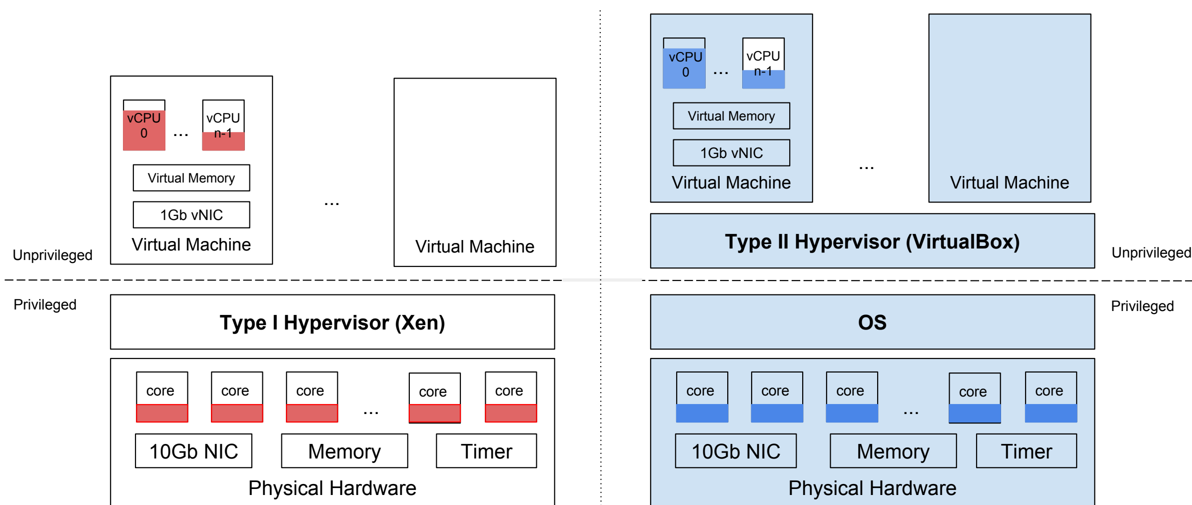


Figure 4.1: Testbed Infrastructures for Type-1 Hypervisor (left) and Type-2 Hypervisor (right)

Testbed Setup. The system used to collect the performance data of our tests consists of several components as shown in Figure 3.2.

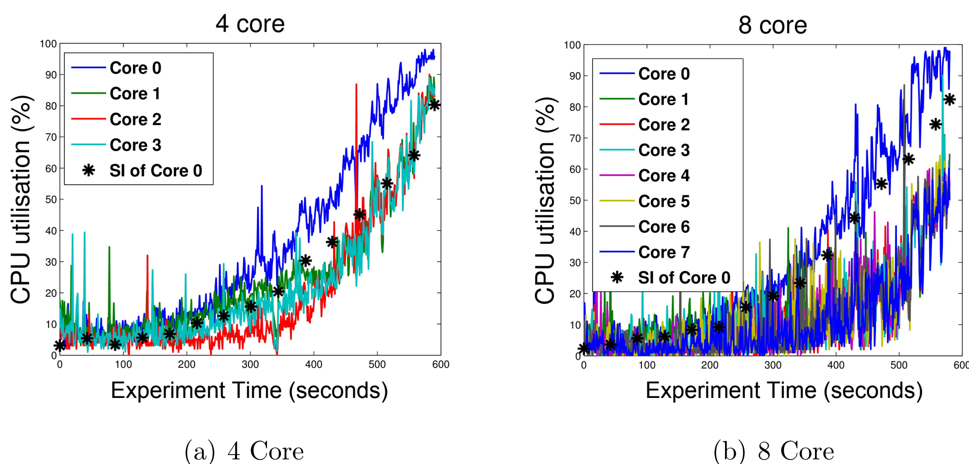
The *data collector* extracts a set of application statistics, e.g. response time and throughput. This is combined with the output of the *data extractor*, which provides hardware characteristics, i.e. utilisation of each core of the VM, memory bandwidth, etc. The data collector can either feed this data directly to the *performance evaluator* or store it a database for future analysis. The performance evaluator is based on the concept of Performance Trees [SBK06, DKS09],

which translate the application and system characteristics into parameters that can be directly used by our performance model. The performance model is then analysed and performance indices of the system are derived and compared to actual measurements. The *automatic controller* optimises the resource configuration for specific performance targets. The system is designed for both on-line and off-line performance evaluation and resource demand estimation, which can be applied in areas such as early stage deployment and run-time management on cloud platforms.

Benchmark. Each server VM is configured with one vCPU with a number of virtual cores (from 1 core up to 8 cores for eight experiments) with 4 GB of memory and one vNIC. To mitigate the effect of physical machine thread switching and to override hypervisor scheduling, each virtual core (vCore) was pinned to an individual physical core. For each experiment, Autobench sends a fixed number of HTTP requests to the server at a specific request rate. The mean request rate incrementally increases for each experiment by 10 req/sec from 50 (e0.02)[4] to 1400 (e0.00071). Figure 4.2 presents the vCore utilisation for the 4 and 8 core VMs running on Virtualbox at increasing request rates for a total duration of 600s. Figure 4.3 shows the corresponding response time and throughput for the VM from 1, 2, 4 and 8 cores. The utilisation, response times, and throughput for the Xen hypervisor are not shown; however, they exhibit similar performance trends.

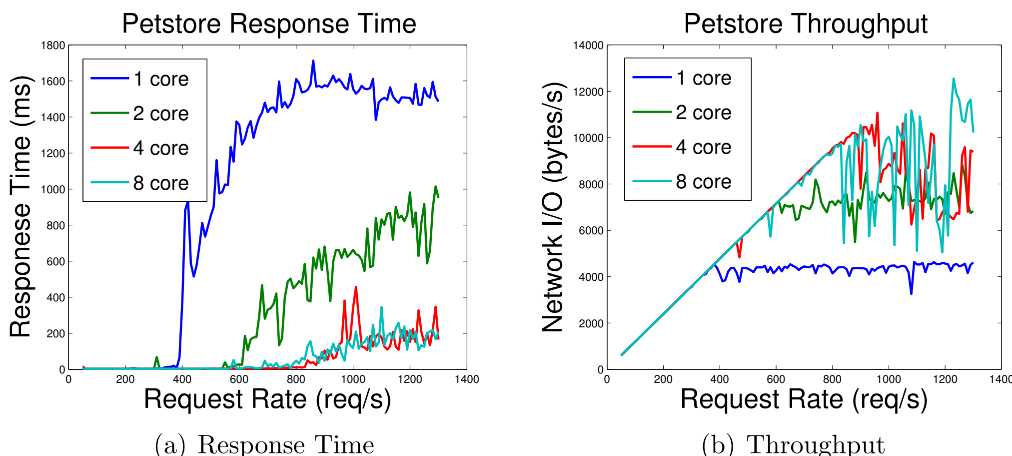
From Figure 4.2(a) and 4.2(b), we observe that the utilisation of vCore 0 reaches 90% and 98% at 500 secs (corresponding to 1200 req/sec) for 4 and 8 vCore servers respectively, while the utilisation of the other vCores are under 80% and 60% for the same setup. Figure 4.3(a) shows that the system becomes overloaded at 400 req/s for a single vCore and at 600 req/s for a dual core. The saturation points for 4 vCores (800 req/s) and 8 vCores (900 req/s) do not reflect the doubling of vCPU capacity. Figure 4.3(b) also shows that for the single and dual core cases, the improvement of system throughput asymptotically flattens with a higher request rate and finally saturates around 4000+ bytes/sec and 7000+ bytes/sec. However, the capacity of the VM servers does not increase linearly when the number of vCores changes from 4 to 8 vCores.

[4]e0.02 refers to an exponential distribution with a mean interarrival time of 0.02s. `http://www.hpl.hp.com/research/linux/httperf/httperf-man-0.9.txt`



(a) 4 Core (b) 8 Core

Figure 4.2: CPU utilisation and software interrupt generated on CPU 0 of 4 core and 8 core VM running the Petstore application on VirtualBox



(a) Response Time (b) Throughput

Figure 4.3: Response time and throughput of 1 to 8 Core VMs running the Petstore application on VirtualBox

When investigating the imbalance of vCore utilisation and lack of scalability across vCores, we have observed that the software interrupt processing causes 90% of the *vCore 0* utilisation, as shown in Figure 4.2. This saturates *vCore 0* as network throughput increases and it becomes the bottleneck of the system. This bottleneck has also been observed in network-intensive web applications executing on non-virtualised multicore servers [HKAC13].

In summary, Figures 4.2 and 4.3 show that, when using the default configurations of VirtualBox, the multicore VM server exhibits poor performance *scalability* across the number of cores for network intensive workloads. Additionally, the *utilisation* of each vCore behaves differently across the cores and as vCore 0 deals with *software interrupts*, it saturates and becomes the

bottleneck of the system.

4.3 Proposed Model

This section describes our proposed model for the performance of a web application running in a multicore virtualised environment. We first give the specification of the model and then present an approximate analytical solution followed by the description of our method to estimate the model parameters. Finally, we validate our model with the testbed from Section 4.2. Here we refer to vCore 0, ..., vCore $n-1$ as CPU 0, ..., CPU $n-1$.

4.3.1 Model Specification

Consider a web application running on an n-core VM with a single NIC (eth0), as in our set-up in Section 4.2.

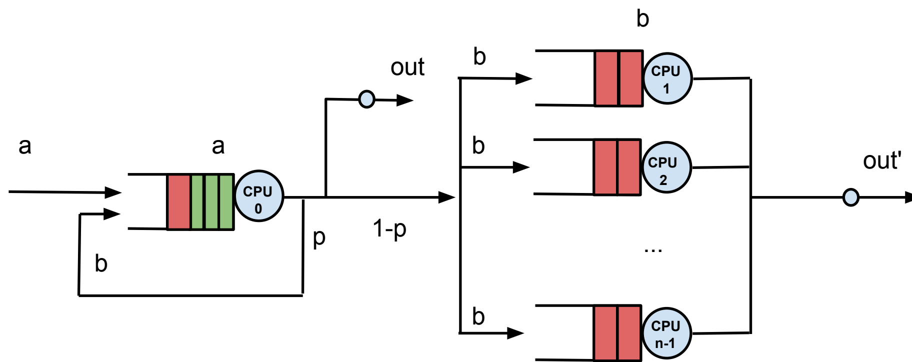


Figure 4.4: Modelling a multicore server using a network of queues

Modelling Multiple Cores: We model the symmetric multicore architecture as we discussed in Section 2.3 using a network of queues. Each queue (CPU 0, ..., CPU $n-1$) in Figure 4.4 represents the corresponding CPU 0, ..., CPU $n-1$ in Figure 2.1. The interrupts generated by eth0 that is represented by job a (green) are handled by CPU 0 by default. In a Linux system, one can see that CPU 0 serves an order of magnitude more interrupts than any other core in */proc/interrupts*. We assume that two classes are served under processor sharing (PS)

queueing discipline in CPU 0; the other queues are M/M/1-PS with single class, which reflects the scheduling policy in most operating systems (e.g. Linux CPU time sharing policy).

When a request arrives from the network (see both Figure 2.1 and Figure 4.4):

1. The NIC driver copies the packet to memory and generates a *interrupt* to signal the kernel that a new packet is readable. This process is represented by job class a (interrupt) being served by CPU 0.

2. The interrupt is processed and the packet is forwarded to the application which reads the request. From the model perspective, a class a job turns into a class b job, which reflects that the interrupt triggers the scheduling of a request process.

3. After the package is pushed down to the appropriate protocol stack layer or application, the jobs (which are represented by job class b in red) are either scheduled to CPU 0 with probability p or to one of the remaining CPUs with probability $1 - p$. Class a and b jobs are served at service rate μ_1 and μ_2 respectively.

4. After a class b job has been processed, the response is sent back to the client. Note that we naturally capture output NIC interrupts by including them into the service time of class a jobs.

In our model, the arrival of jobs is a Poisson process with arrival rate λ and job service times are exponentially distributed. The system has a maximum number of jobs that it can process as shown in Figure 4.3, which is also very common for computer systems. For each experiment, an arrival is dropped by the system if the total number of jobs in the system has reached a specified maximum value N.

The preemptive multitasking scheme of an operating system, such as Windows NT, Linux 2.6, Solaris 2.0 etc., utilises the interrupt mechanism, which suspends the currently executing process and invokes the kernel scheduler to reschedule the interrupted process to another core. Otherwise, when a class a job arrives, a class b job executing in CPU 0 could be blocked. However, in a multicore architecture, the blocked processes could experience a timely return

to execution by a completely fair scheduler, shortest remaining time scheduler, or some other CPU load-balancing mechanism. To simplify the model, class a and class b jobs are processed separately with a processor sharing policy in CPU 0.

4.3.2 CPU 0

The proposed queueing model in Figure 4.4 abstracts the process of serving web requests on a multicore architecture. In this model, CPU 1 to CPU $n-1$ are modelled as standard M/M/1-PS queues, the arrivals to which emanate at CPU 0 as class b jobs. An M/M/1-PS queue is one of the common queue types in the literature [HP92]. The nontrivial part of the model, however, is CPU 0. CPU 0 processes two classes of jobs, a and b, and the number of jobs can be described as a two dimensional Markov chain $X = (i, j)$, where i is the number of class a job and j is the number of class b job. Figure 4.5 illustrates the state transitions corresponding to the generator matrix of its stochastic process, $\mathbf{Q}$.

One can compute the stationary distribution numerically by solving the normalised left zero eigenvector of $\mathbf{Q}$. However, as the capacity of the system, N, is a very large number in the real system, the size of $\mathbf{Q}$, is combinatorially large and hence, computing the zero eigenvector becomes infeasible. Next, we obtain the stationary distribution of the Markov chain.

4.3.3 Two-class Markov Chain and its Stationary Distribution of CPU 0

The model specification given in Section 4.3.1 and the state transition diagram of Figure 4.5 make the approximating assumption that the total service rate for each class (a and b) does not degrade as the population of the network increases, remaining at the constant values μ_1 and μ_2. Therefore the classes behave independently and the modelled behaviour of CPU 0 is equivalent to a tandem pair of single-class PS queues with rates μ_1 (for class a) and μ_2 (for class b) respectively. The arrival rate at the first queue is λ and at the second $p\lambda$ (since we are considering only CPU 0). This is a standard BCMP network [HP92] with a population

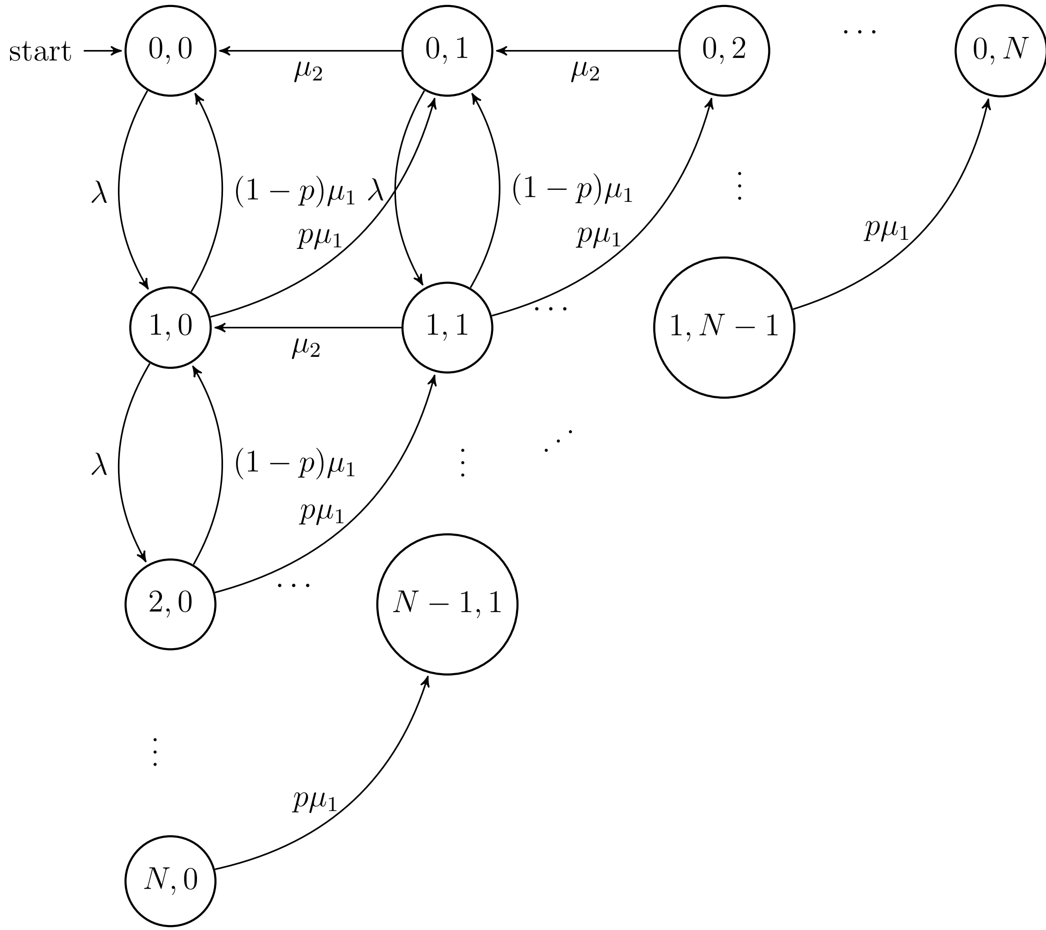


Figure 4.5: State transition diagram for CPU 0

constraint and so has the product-form given in equation (4.2)[5]. Moreover, the result is a trivial application of the Reversed Compound Agent Theorem (RCAT), see for example [Har03] [HLP09]. The normalising constant can be obtained as a double sum of finite geometric series and gives the value of $\pi_{0,0}$ shown in equation (4.1).

We therefore have the following product-form solution:

Proposition 1. *Assuming that a steady state exists, let the steady-state probability of state (i, j) in Figure 4.5 be denoted $\pi_{i,j}$. Then,*

$$\pi_{0,0} = \frac{(\alpha - 1)(\alpha - \beta)(\beta - 1)}{\alpha^{N+2}(\beta - 1) + \beta^{N+2}(1 - \alpha) + \alpha - \beta}, \tag{4.1}$$

[5]We thank a referee for pointing out that the result was first derived in [Lam77]

and

$$\pi_{i,j} = \alpha^i \beta^j \pi_{0,0}, \tag{4.2}$$

where

$$\alpha := \frac{\lambda}{\mu_1} \quad \textit{and} \quad \beta := \frac{p\lambda}{\mu_2}. \tag{4.3}$$

Proof. By the proceeding argument, the BCMP Theorem yields,

$$\pi_{i,j} = C\pi_1(i)\pi_2(j).$$

where C is a normalising constant. The marginal probabilities are,

$$\pi_1(k) = \alpha^k \pi_1(0), \pi_2(k) = \beta^k \pi_2(0) \quad \forall k = 0, 1, \ldots, N.$$

Therefore,

$$\pi_{i,j} = C\pi_1(i)\pi_2(j) = C\alpha^i\beta^j\pi_1(0)\pi_2(0) = \alpha^i\beta^j\pi_{0,0}.$$

Normalising, we have

$$\sum_{i,j} \pi_{i,j} = 1$$
$$\sum_{i,j} \alpha^i\beta^j\pi_{0,0} = 1$$
$$\pi_{0,0} \sum_{i=0}^{N} \sum_{j=0}^{N-i} \alpha^i\beta^j = 1$$

Since

$$\sum_{i=0}^{N} \sum_{j=0}^{N-i} \alpha^i\beta^j = \frac{\alpha^{N+2}(\beta-1) + \beta^{N+2}(1-\alpha) + \alpha - \beta}{(\alpha-1)(\alpha-\beta)(\beta-1)},$$

we obtain

$$\pi_{0,0} = \frac{(\alpha-1)(\alpha-\beta)(\beta-1)}{\alpha^{N+2}(\beta-1) + \beta^{N+2}(1-\alpha) + \alpha - \beta}.$$

□

4.3.4 Average Sojourn Time of CPU 0

Proposition 1 provides the stationary distribution of the Markov chain associated with CPU 0. With that information, we can find the average number of jobs in the system.

Proposition 2. *Let the random variable k denote the total number of jobs at CPU 0. Then,*

$$E(k) = \frac{g(\alpha, \beta) - g(\beta, \alpha) + (\beta - \alpha)(2\alpha\beta - \alpha - \beta)}{[\alpha^{N+2}(\beta - 1) + \beta^{N+2}(1 - \alpha) + \alpha - \beta](\alpha - 1)(\beta - 1)}, \tag{4.4}$$

where $g(x, y) := x^{N+2}(y - 1)^2(xN - N - 1)$.

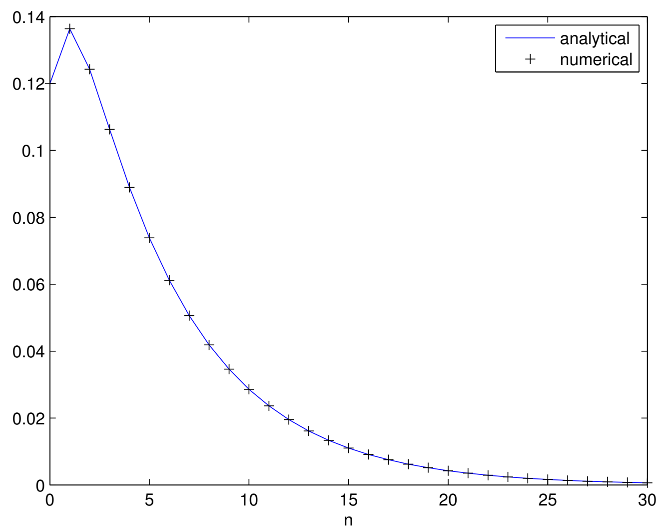


Figure 4.6: Comparing numerical and analytical solution of $E(k)$

Proof. By definition, the expected number of jobs is

$$E(k) = \sum_{i,j} (i + j)\pi_{i,j}.$$

Using results from Proposition 1, we have

$$\begin{aligned}
E(k) &= \sum_{i,j}(i+j)\pi_{i,j}, \\
&= \pi_{0,0}\sum_{i,j}(i+j)\alpha^i\beta^j, \\
&= \pi_{0,0}\sum_{i=0}^{N}\sum_{j=0}^{N-i}(i+j)\alpha^i\beta^j, \\
&= \pi_{0,0}\frac{g(\alpha,\beta)-g(\beta,\alpha)+(\beta-\alpha)(2\alpha\beta-\alpha-\beta)}{(\alpha-1)^2(\alpha-\beta)(\beta-1)^2}, \\
&= \frac{g(\alpha,\beta)-g(\beta,\alpha)+(\beta-\alpha)(2\alpha\beta-\alpha-\beta)}{[\alpha^{N+2}(\beta-1)+\beta^{N+2}(1-\alpha)+\alpha-\beta](\alpha-1)(\beta-1)},
\end{aligned}$$

where $g(x,y) := x^{N+2}(y-1)^2(xN-N-1)$. □

Figure 4.6 plots the value of $E(k)$ against N.

Consider again CPU 0 with two job classes a and b. Arrivals will be blocked if the total number of jobs reaches N. The probability function of the total number of jobs at CPU 0 can be calculated as,

$$P_N = P[n_a+n_b=N] = \sum_{i,j}^{i+j=N}\pi_{i,j}$$

Using Proposition 2, a job's expected sojourn time $E(T)$ can be calculated from the long-term average effective arrival rate λ and the average number of jobs $E(k)$, using *Little's Law* for the system as follows:

$$E(T) = \frac{E(k)}{\lambda(1-P_N)}$$

4.3.5 Average Service Time and Utilisation of CPU 0

Proposition 3. *Let T_s be the random variable denoting the service time of a job γ entering service. The expected service time is*

$$E(T_s) = \frac{1}{\mu_1} n_a^0 + \frac{1}{\mu_1} \frac{\lambda}{\lambda + p\mu_1}(1 - n_a^0) + \frac{1}{\mu_2} \frac{p\mu_1}{\lambda + p\mu_1}(1 - n_a^0), \tag{4.5}$$

where

$$n_a^0 = \pi_{0,0} \frac{1 - \beta^{N+1}}{1 - \beta}.$$

Proof. Let n_a be the current number of class a job in the system, we have

$$\begin{aligned}
E(T_s) &= E(T_s \mid \gamma \text{ is job a}) P(\gamma \text{ is job a}) \\
&\quad + E(T_s \mid \gamma \text{ is job b}) P(\gamma \text{ is job b}) \\
&= \frac{1}{\mu_1} P(\gamma \text{ is job a}) + \frac{1}{\mu_2} P(\gamma \text{ is job b}) \\
&= \frac{1}{\mu_1} P(\gamma \text{ is job a} \mid n_a = 0) P(n_a = 0) \\
&\quad + \frac{1}{\mu_1} P(\gamma \text{ is job a} \mid n_a > 0) P(n_a > 0) \\
&\quad + \frac{1}{\mu_2} P(\gamma \text{ is job b} \mid n_a = 0) P(n_a = 0) \\
&\quad + \frac{1}{\mu_2} P(\gamma \text{ is job b} \mid n_a > 0) P(n_a > 0).
\end{aligned}$$

Since

$$P(\gamma \text{ is job b} | n_a = 0) = 0, \qquad P(\gamma \text{ is job a} | n_a = 0) = 1,$$

we have

$$\begin{aligned}
E(T_s) &= \frac{1}{\mu_1}P(n_a = 0) \\
&\quad + \frac{1}{\mu_1}P(\gamma \text{ is job a} \mid n_a > 0)P(n_a > 0) \\
&\quad + \frac{1}{\mu_2}P(\gamma \text{ is job b} \mid n_a > 0)P(n_a > 0). \\
&= \frac{1}{\mu_1}P(n_a = 0) + \frac{1}{\mu_1}\frac{\lambda}{\lambda + p\mu_1}P(n_a > 0) \\
&\quad + \frac{1}{\mu_2}\frac{p\mu_1}{\lambda + p\mu_1}P(n_a > 0) \\
&= \frac{1}{\mu_1}P(n_a = 0) + \frac{1}{\mu_1}\frac{\lambda}{\lambda + p\mu_1}(1 - P(n_a = 0)) \\
&\quad + \frac{1}{\mu_2}\frac{p\mu_1}{\lambda + p\mu_1}(1 - P(n_a = 0)).
\end{aligned}$$

Notice that from previous results,

$$\begin{aligned}
P(n_a = 0) &= \sum_{j=0}^{N} \pi_{0,j} \\
&= \pi_{0,0} \sum_{j=0}^{N} \alpha^0 \beta^j \\
&= \pi_{0,0} \frac{1 - \beta^{N+1}}{1 - \beta}.
\end{aligned}$$

Therefore,

$$E(T_s) = \frac{1}{\mu_1}n_a^0 + \frac{1}{\mu_1}\frac{\lambda}{\lambda + p\mu_1}(1 - n_a^0) + \frac{1}{\mu_2}\frac{p\mu_1}{\lambda + p\mu_1}(1 - n_a^0),$$

where

$$n_a^0 = \pi_{0,0}\frac{1 - \beta^{N+1}}{1 - \beta}.$$

□

With the result above, the utilisation of a single core can be derived by the *Utilisation Law*,

$$U = \lambda E(T_s)$$

4.3.6 Likelihood for Estimating Parameters

The stationary distribution π of the Markov process in Figure 4.5 with generator matrix $\mathbf{Q}$ and the expected number of jobs $E(k)$ are given in Propositions 1 and 2. There are three corresponding parameters, μ_1, μ_2, and p. We assume that the average response time for a certain request arrival rate λ_i can be estimated from real system measurements. From our previous observations, for example, when a one core system receives 100 req/sec, on average, 2.9% of the CPU utilisation are spent for processing software interrupts while for 200 req/sec, this amount increases to 7.2%. We can obtain μ_1 from utilisation law,

$$\frac{\bar{\lambda}}{\mu_1} = \bar{U}_{si} \tag{4.6}$$

where $\bar{U}_{si}$ denotes the average utilisation of software interrupts (si) processed by CPU 0 during a monitoring window of size t and $\bar{\lambda}$ is the average λ_i during t. Then the reciprocal of μ_1 is the mean service time of CPU 0 handling si. Note that by using the average utilisation for software interrupts to calculate μ_1, the service time for a class a job includes the service time for *all* software interrupts involved to successfully process the corresponding class b job (see Section 4.3.1). Here, we find μ_1 to be 3301 req/sec. In the single core case, p is 1. However, for multiple core cases, p can be obtained by the inverse proportion of the utilisation as a load balancing across multiple cores.

Let T_i be the average response time estimated for a certain arrival rate from the model and T_i' be the average time from the real system measurements when the arrival rate is $\lambda_i, i = 1, \ldots, m$. Since the estimated response time T' is the mean of samples, it is approximately a normally distributed random variable with mean T and variance $\frac{\sigma_T^2}{n}$ when the number of samples n is very large [CANK03]. Hence μ_2 can be estimated by maximising the log-likelihood function,

$$\log \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma_i^2/n_i}} \exp\left[\frac{(T'_i - T_i)^2}{2\pi\sigma_i^2/n_i}\right] \tag{4.7}$$

Maximising the log-likelihood function above is equivalent to minimising the weighted sum of squared errors:

$$\sum_{i=1}^{m} \frac{(T'_i - T_i)^2}{2\pi\sigma_i^2/n_i} \tag{4.8}$$

Now the problem of finding the parameters becomes an optimisation problem,

$$\mu_2 = \arg\min_{\mu_2} \sum_{i=1}^{m} \frac{(T'_i - T_i)^2}{2\pi\sigma_i^2/n_i} \tag{4.9}$$

The optimisation problem can be solved in different ways, such as steepest descent and truncated Newton [CANK03]. We carried out the experiments in the single core case with λ varying from 10 req/s to 500 req/s. For each λ we sent requests from 300 to 30 000 req/s and measured the mean response time and the corresponding standard deviation.

4.3.7 Combined Model

In the previous section, we analysed the properties of CPU 0, which gives us a better understanding of how its performance is affected by interrupts. To build the entire model, we will combine the previous results of CPU 0 and the results of CPU 1 to CPU $n-1$ given in [HP92].

For K jobs arriving in the system, we expect Kp of them will stay in CPU 0 and $K(1-p)$ of them will be sent to CPU 1, ..., CPU $n-1$. Given request arrival rate λ, we approximate the arrival rate of jobs at CPU 1, ..., CPU $n-1$ as $\lambda(1-p)$. We further assume that those jobs are uniformly assigned to different cores and so for CPU i, the corresponding (class b) job arrival rate is $\lambda_i = \lambda(1-p)/(n-1)$. Given the service rate of class b jobs is μ_2, the expected number of jobs at these CPUs is $\lambda_i/(\mu_2 - \lambda_i)$, $\forall i = 1, \ldots, n-1$.

Table 4.1 gives the brief summary of key model parameters. Let k_i denote the number of jobs in the queue of CPU i; then by Little's Law, the expected sojourn time of a request in the

	CPU 0	*CPU 1, ..., CPU n-1*
Arrival Rate	λ $p\mu_1$(a $\rightarrow$ b)	$\lambda(1-p)$
Service Rate	μ_1 (a) μ_2 (b)	μ_2
Mean Jobs	Proposition 2	$\lambda_i/(\mu_2 - \lambda_i)$

Table 4.1: Summary of the key parameters in the multicore performance prediction model

Values of μ_2 (req/sec)			
1 core	2 core	4 core	8 core
367	345	300	277

Table 4.2: Likelihood estimation of the mean service rate (req/sec) for class b job

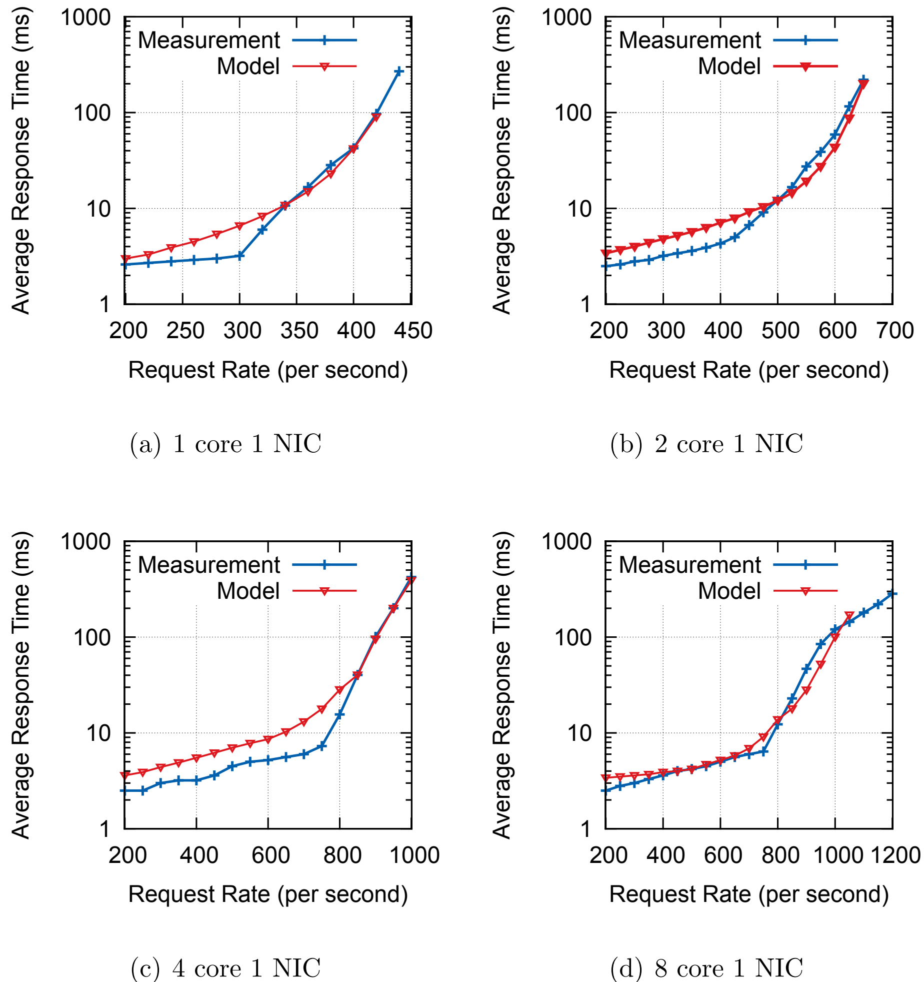
whole system is,

$$\begin{aligned} E(T_{\text{sys}}) &\approx \frac{E(k_0 + k_1 + \cdots + k_{n-1})}{\lambda} \\ &= \frac{E(k_0) + E(k_1) + \cdots + E(k_{n-1})}{\lambda}. \end{aligned}$$

4.3.8 Validation

We validate our model against real system measurements of response time and throughput, focusing on benchmarks running on the VirtualBox hypervisor and using the system set-up of Section 4.2.

Prior to validation, we conducted baseline runs of the benchmark in our test-bed system. Each measurement point was the average over 200 measurements. We assume this is long enough to acquire a steady state average response time for each request. For each run, we varied the number of cores and collected information about workload and response time for the parameter estimation (see Section 4.3.6). The parameters we obtained for class b decrease from 1 core to 8 cores as shown in Table 4.2. The decreasing μ_2 captures the fact that the web server scales poorly on multiple cores because of (i) the virtualisation overhead; (ii) the inherent problem of multicore, such as context switching overhead. Figure 4.7 shows the validation of response time.



(a) 1 core 1 NIC

(b) 2 core 1 NIC

(c) 4 core 1 NIC

(d) 8 core 1 NIC

Figure 4.7: Response time validation of 1 to 8 core with 1 NIC

4.4 Scalability and Model Enhancement

In this section, we first describe a set of complementary techniques of system and hardware configurations aimed to prevent the single core bottleneck discussed in Section 4.2. We apply one of the techniques to increase parallelism and improve performance for multicore web applications. Second, we derive our model for performance under an improved configuration. We then validate our model under the new configurations and show that the results fit with the obtained performance improvements.

4.4.1 Scalability Enhancement

Multiple network interfaces can provide high network bandwidth and high availability [JJ09, HKAC13, VF07]. Enforcing CPU affinity for interrupts and network processing has been shown to be beneficial for SMP systems [VF07] and the same benefits should apply to virtualised multicore systems. Combining multiple NICs and CPU affinity allows us to distribute the software interrupts for different NICs to different cores and hence mitigate load imbalance. In real systems, installing multiple network interfaces might cause space and power issues; however, in virtualised environments, this can be trivially achieved by using virtual NICs. To gain the benefit of reducing the bottleneck of CPU 0, as we illustrated in Figure 4.1, the network resource and the physical CPU resource are over-provisioned to ensure that these two resources are not the bottlenecks in our testbed. For our enhanced configuration, we configure multiple vNICs as follows:

- Fix the number of web server threads to the number of cores and assign each web server thread to a dedicated core to avoid the context switching overhead between two or more threads [HKAC13].
- Distribute the NIC interrupts to multiple cores by assigning multiple virtual NICs, i.e. vboxnet, to the VM.

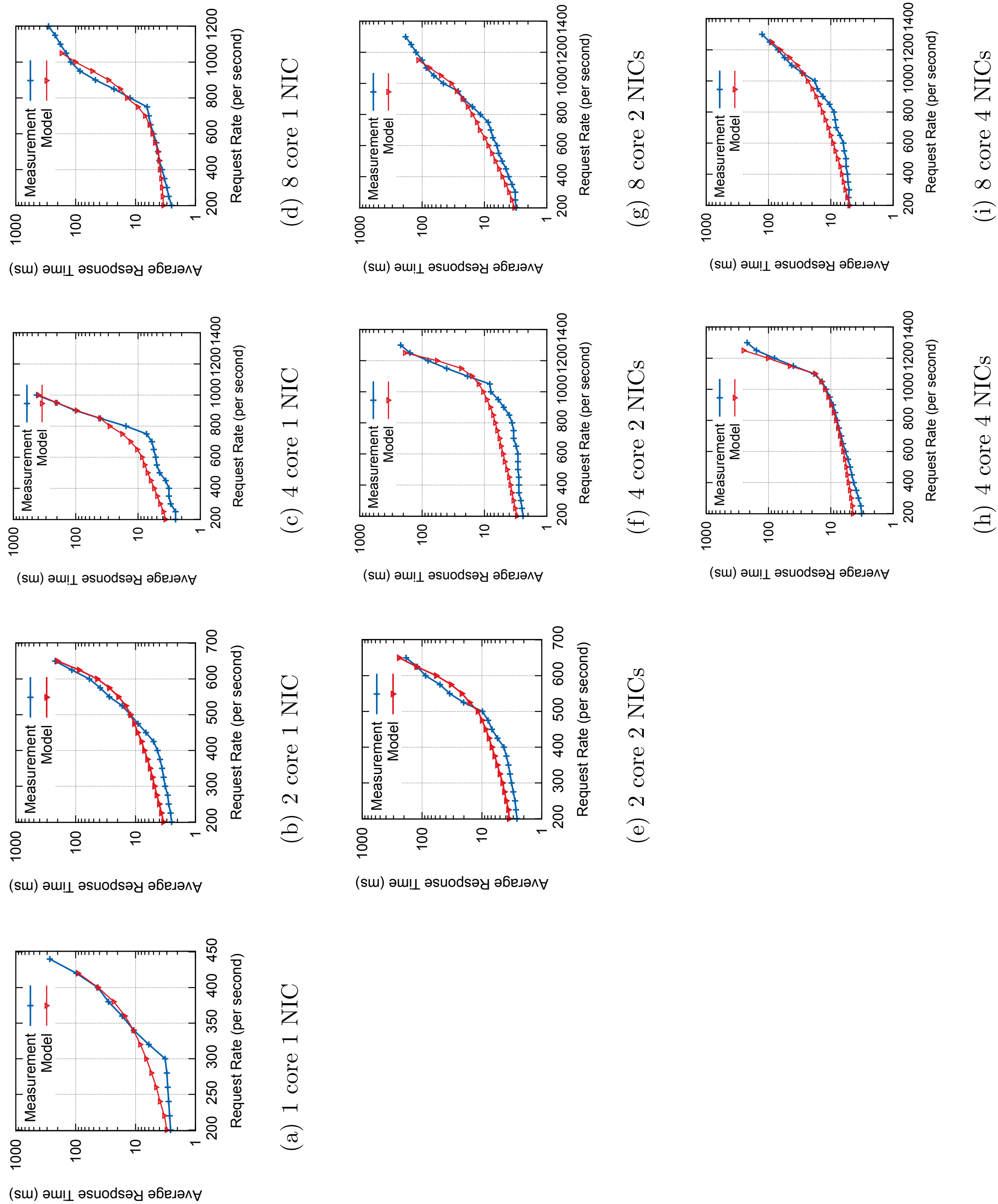
4.4.2 Model Enhancement

Since we model the imbalance of multicore system by distinguishing two different types of queues, we can derive the model for the new configuration by increasing the number of leading two-class queues to match the number of cores m which deal with NIC interrupts. Recall that our baseline model assumes a single core (queue) handling NIC interrupts (job a). Consider the situation when job a comes to m two-class queues (equals to m CPU 0), in which m represents the number of cores that handle NIC interrupts. Then, a class a job transfers into a class b job and either returns to the queue with probability p or proceeds to CPU m, …, CPU $n-1$ with probability $1-p$.

4.4.3 Prediction with Previous Parameters

We apply the model for the enhanced configurations with the same parameters as shown in Table 4.2. What we expect to see is that the performance of the application improves with the new configurations, and exhibits better scalability. This is due to: (i) network interrupt storms are distributed to more cores instead of directing to a single core; (ii) the use of flow affinity [HKAC13, VF07], which ensures that all packets in a TCP flow (connection-oriented) are processed by a single core. This reduces contention for shared resources, minimises software synchronization and enhances cache efficiency [JJ09].

The prediction results with previous parameters are shown in Figure 4.8. As we can see that, with 4 cores and 1 NIC, the knee-bend in system performance occurs at around 800 req/sec; using 2 NICs this increases to around 1000 req/sec and for 4 NICs to around 1200 req/sec. These figures suggest that more virtual NICs can help improve TCP/IP processing in multicore environments. The summary of the error found in all validation results of Figure 4.7 and Figure 4.8 are shown in Table 4.3. The average relative modelling error is around 15%. This shows a tendency to decrease with an increasing number of NICs. We see a relative error of e.g. 7.9% and 7.4%, for a 4 core machine with 2 NICs and a 4 core machine with 4 NICs, respectively. Since distributing the NIC interrupts in the real system causes extra context switching overhead, the response time of relatively low intensity workloads (i.e. 200 to 600 req/sec) is round 10-20% higher than that for the default configuration.



(a) 1 core 1 NIC

(b) 2 core 1 NIC

(c) 4 core 1 NIC

(d) 8 core 1 NIC

(e) 2 core 2 NICs

(f) 4 core 2 NICs

(g) 8 core 2 NICs

(h) 4 core 4 NICs

(i) 8 core 4 NICs

Figure 4.8: Revalidation of response time of 1 to 8 core with multiple number of NICs

Table 4.3: Relative errors between model and measurements (%)

	1 NIC				2 NICs			4 NICs		Overall
Num. of Core	1	2	4	8	2	4	8	4	8	
Response Time	23.8	23.2	25.8	11.3	19.4	7.9	10.3	7.4	14.2	15.9
Throughput	14.1	12.9	13.4	16.5	14.5	11.9	15.6	10.6	16.7	14.02
Util. Core 0 to m-1	10.5	7.9	8.4	9.8	8.4	8.9	12.9	11.4	13.4	10.2
Util. Core m to N-1	-	-9.4	-14.6	-23.7	-	-10.4	- 16.7	-	-17.8	-15.4

4.4.4 Prediction validation for Type I hypervisor – Xen

See Appendix A.

4.4.5 Model Limitations

We identify several factors that affect model accuracy:

1. The *routing probability* p: we use a simple load balancing policy as we discussed in Section 4.3.7, which cannot represent the Linux kernel scheduling algorithm used in our testbed, which is a completely fair scheduler. More advanced scheduling policies like O^2 [WGB+10] can also not be described with this simple model.

2. *Interrupt priority*: in general, NIC interrupts (job a) have higher priority than system and user processes (job b). In the single core case, job b is blocked when a new job a arrives. However, in the multicore case, the scheduler will assign it to another core. To simplify the model, we do not consider priorities and interference between job classes a and b.

3. *Context switching overhead*: an operating system executes a context switch by loading a new context, i.e. registers, memory mappings, etc., in one CPU. Though we try to reduce the context switching overhead by assigning each web server thread statically to a distinct core, other context switches, such as register, task, and stack, need to be considered.

4. *Hypervisor overhead*: our model implicitly considers virtualisation overhead, e.g. via the decrease of service rate with increasing number of cores. However, how the overhead of processing requests at the different virtualisation layers has yet to be accounted for.

4.5 Summary

This chapter has presented a performance model for web applications deployed in multicore virtualised environments. The model is general enough to capture the performance of web

applications deployed on multicore VMs and can account for hardware idiosyncrasies such as CPU bottlenecks and interrupt influences. We gave an approximate analytical solution and validated our model in our testbed using an open-source web application running on multicore VMs. In addition, we presented a simple approach to achieve better scalability for multicore web servers through use of virtual hardware. We also demonstrated the applicability of our model in the enhanced configurations. In the next chapter, we will refine our method to overcome the approach limitations, mainly focused on the hypervisor overhead.

Chapter 5

Diagnosing and Managing Performance Interference in Multi-Tenant Clouds

5.1 Introduction

Virtual machine consolidation is attractive in cloud computing platforms for several reasons including reduced infrastructure costs, lower energy consumption and ease of management. However, the interference between co-resident workloads caused by virtualisation can violate the SLOs that the cloud platform guarantees. Existing solutions to minimise interference between VMs are mostly based on comprehensive micro-benchmarks or online training which makes them computationally intensive. In this chapter, we present CloudScope, a system for diagnosing interference for multi-tenant cloud systems in a lightweight way.

The demand for cloud computing has been constantly increasing during recent years. Millions of servers are hosted and utilised in data centres every day and many organisations deploy their own, private cloud services to be able to better manage their own computing infrastructure [DK13]. Virtualisation enables cloud providers to efficiently allocate resources to tenants on demand and consolidate tenants' workloads to reduce operational cost. Successful management of a cloud platform requires the optimal assignment of incoming VMs or *guests* to available PMs or *hosts*. This scheduling problem is constrained by both the tenants' SLOs and

the available resources. Co-resident VMs are desirable for the cloud provider as this means utilising available resources more efficiently. However, the more VMs are consolidated on a single machine, the more instances compete for resources and the hypervisor capacity. As a result of this *interference*, guest systems may experience high performance variations which lead to unpredictable system behaviour and SLO violations [XBNJ13, CH11] such as a drop in application throughput or an increase in the response time of a web service.

Recognising this problem, researchers have developed many methods to identify and predict performance interference. This work can be categorised into two groups: (1) machine learning-based approaches [DK13, CH11, NKG10, RKG+13, CSG13, KEY13, ZT12, YHJ+10] and (2) queueing model-based approaches [NBKR13, CKK11, KKEY12]. The first group uses sophisticated micro-benchmarks and online training to predict the performance interference of different applications. As prediction is based on historical data, adaptation to unknown workloads becomes difficult. Also, continuously updating the models is computationally expensive. The second group relies on unified queueing models and system attributes such as service and arrival rates which are usually difficult to obtain due to system complexity and varying workloads. In addition, these methods support only specific hardware configurations and existing applications. They do not provide a method which is general and efficient enough for complex cloud environments where applications change frequently.

In this chapter, we consider this co-residency problem and present CloudScope, a system that diagnoses the bottlenecks of co-resident VMs and mitigates their interference based on a lightweight prediction model. The model is a discrete-time Markov chain that predicts performance slowdown when compared to an environment without hypervisor overhead and resource contention, represented by a *virtualisation-slowdown* (V-slowdown) factor. The key feature of CloudScope is its ability to efficiently characterise the performance degradation by probing the system behaviour which includes both the behaviour of the hypervisor layer and the hardware characteristics of different workloads. Current systems [DK13, CH11, RKG+13, CSG13, ZT12] do not explicitly consider these factors in an analytical model.

CloudScope's model parameters can be easily obtained via hypervisor profiling utilities such as

`xentop`. As these values are by default reported from the hypervisor, no overhead is introduced. CloudScope employs its model to control guest instance placement and reduce SLO violations by minimising interference effects. It also manipulates the hypervisor to achieve an optimal configuration by, for example, increasing the CPU share available to the Dom0. Because of the practicality of this analytical model, adaptive hypervisor control and migration or consolidation of VMs becomes a lightweight and fast operation that does not require complex training or micro-benchmarking.

We implement CloudScope and evaluate its accuracy and effectiveness using a wide spectrum of workload scenarios including a set of CPU, disk, and network intensive benchmarks and a real workload using Hadoop MapReduce. CloudScope's interference prediction model can achieve a minimum prediction error of 4.8% and is less than 20% for all our test workloads. We illustrate the feasibility of CloudScope's interference-aware VM scheduler by comparing it to the default scheduler of a CloudStack deployment and achieve an overall performance improvement of up to 10%. In addition we show how CloudScope can be applied to self-adaptive hypervisor control to answer questions such as: which configurations can best serve the guests performance requirements? We make the following contributions:

- We introduce a lightweight analytical model solely based on available system parameters to predict the impact of co-residing VMs on performance.
- We combine both the behaviour of the hypervisor and the specific hardware requirements of different workloads in our model for fast and accurate predictions.
- We implement an interference-aware scheduler for a CloudStack deployment and illustrate the effect of a self-adaptive Xen control domain.

The rest of this chapter is organised as follows. Section 5.2 demonstrates the performance degradation caused by hypervisor overhead and resource contention using benchmarking experiments. We present CloudScope's system design and its performance model in Section 5.3 and discuss details on interference handling in Section 5.4. The validation and experimental results are presented in Section 5.5. Section 5.6 concludes this chapter.

5.2 Characterising Performance Interference

Performance interference in multi-tenant data centres is well studied both in the context of prediction and measurement. Many benchmarking studies are devoted to understanding the performance of EC2 VM instances [DK13, TIIN10], their network [XBNJ13, RKG+13, SWWL14] and applications deployed in them [BRX13, CHO+14]. These studies found that virtualisation and multi-tenancy are the major causes for resource contention as multiple VMs are placed on a single host. This leads to performance variation in EC2 instances. In this section, we will quickly recap the background of virtualisation and show that these performance bottlenecks depend on different workload parameters and quantify the impact on the underlying system.

5.2.1 Recapping Xen Virtualisation Background

The Xen hypervisor is widely used as the basis of many commercial and open source applications. It is also used in the largest clouds in production such as Amazon and Rackspace. We picked Xen as the basis for our model in this work as it is the only open source bare-metal hypervisor. Although PV has significant performance benefits as demonstrated in the original Xen paper [BDF+03], the existence of an additional layer between the VM applications and the hardware introduces overhead. The overhead depends on the type of workload that is executed by the guest [GCGV06]. In general, CPU-intensive guest code runs close to 100% native speed, while I/O might take considerably longer due to the virtualisation layer [GCGV06]. However, CPU oversubscription is common in cloud environments which also limits the performance of CPU-intensive jobs [BWT12] . CPU performance is affected by the time slices allocated to the VMs which are based on a weight (the CPU share for each VM and the hypervisor), a cap (the maximum usage), and the amount of pending tasks for the physical processors [CG05, CGV07, PLM+13]. Recall what we discussed in Section 2.4, disk and network I/O suffer from overhead caused by moving data between the VM, shared memory, and the physical devices [SWWL14, PLM+13]. Xen uses two daemon processes, `blkfront` and `blkback`, to pass I/O requests between DomU and Dom0 via a shared memory page. Similarly,

PM	Two Intel 8-core 2.9 GHz (32 hyper-threading), 256 GB Memory
VM-CPU	4 vCPUs (1 GHz per vCPU), 2 GB Memory, 5 GB local storage
VM-disk	4 vCPUs (1 GHz per vCPU), 8 GB Memory, 32 GB local storage
VM-net	4 vCPUs (1 GHz per vCPU), 8 GB Memory, 1 vNic

Table 5.1: Benchmarking Configuration

`netfront` and `netback` share two pages, one for passing packets coming from the network and the other for packets coming from DomU as shown in Figure 2.3. Xen also places upper limits on the number of I/O requests that can be placed in this shared memory which may result in delays due to blocking. This means that not only the latency increases but also that the bandwidth is reduced.

5.2.2 Measuring the Effect of Performance Interference

To illustrate the problems resulting from the above described virtualisation techniques, we measure the performance of CPU, disk, and network intensive applications sharing resources on one physical server. Table 5.1 gives the configurations for the PM and the VMs for each experiment.

To generate a CPU intensive workload we use the `sysbench`[1] benchmark with 4 threads on 4 virtual CPU cores (one thread on each virtual core) to generate CPU load average for a single VM. We measure the load average of the VM for different prime number calculations. For prime numbers up to 2000, 3000, and 5000, we see load average of 25%, 50%, and 90% respectively over a period of time within a single VM. Figure 5.1 explains why different prime numbers produce different load averages on single VM. This measurement includes job process setup and termination overhead including both idle and busy periods during the experiment. After the experiment of single VM running these three prime numbers, we used the same VM image to launch up to 45 VMs to concurrently run these prime number to see the effect of performance interference on CPU resource. What we expect to see here is an increase in job completion time due to resource over-subscription and hypervisor scheduling overhead.

[1]Sysbench Benchmark. `http://wiki.gentoo.org/wiki/Sysbench`

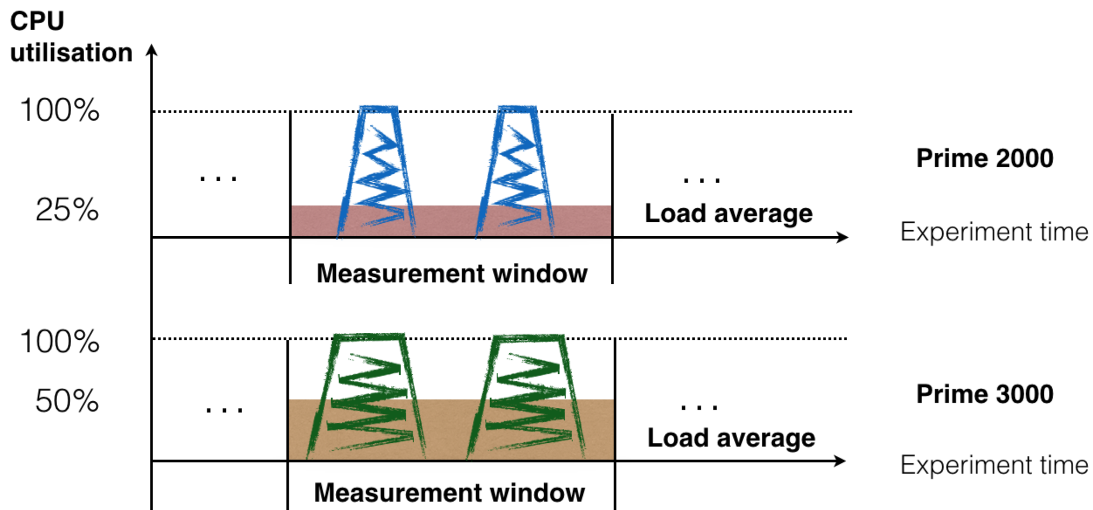
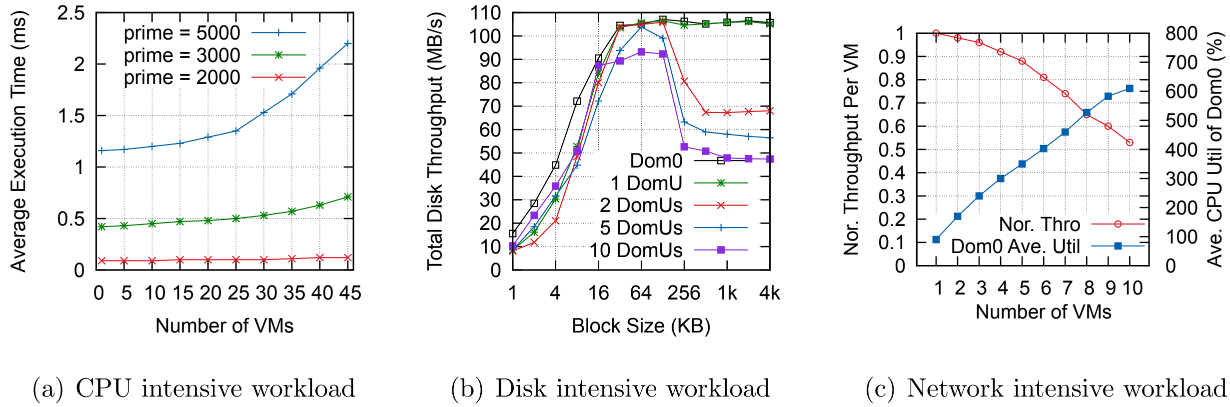


Figure 5.1: The load average (utilisation) example of the VM running sysbench experiment

Figure 5.2(a) shows the mean completion time for the jobs while increasing the number of co-resident VMs from 1 to 45. We observe that the time for calculating prime numbers up to 2000 is stable until 35 VMs and then slightly increases. For prime numbers up to 3000 and 5000, we can observe an increase in completion time for 5 co-resident VMs with a steeper increase from 25 VMs on. This behaviour reflects the effects of (1) the high VM CPU load, (2) the Xen scheduling overhead and (3) CPU resource contention. Resource contention depends on the individual loads within each VM and the number of VMs running such loads. For example, when each VM runs prime 3000 it will produce an average CPU load of 50% individually on each vCPU; thus we expect the physical machine to be saturated with 46 simultaneously running VMs:

$$VMs_{50} = \frac{32 \times 2.9 \text{ GHz}}{4 \times 1 \text{ GHz} \times 50\%} \approx 46$$

Hence, the results for prime 3000 in Figure 5.2(a) show the Xen scheduling overhead without CPU contention which can go as high as 1.7x for 45 VMs. We also observe that for prime 5000, the mean execution time is affected earlier as each VM produces 90% utilisation. After 25 VMs, the increase in execution time then comes from both Xen scheduling overhead and resource contention.



(a) CPU intensive workload (b) Disk intensive workload (c) Network intensive workload

Figure 5.2: Co-resident VM performance measurements for (a) CPU, (b) disk and (c) network intensive workloads for revealing different system bottlenecks. Figure (a) shows the average execution time of executing prime 2000, 3000, and 5000. Figure (b) shows the total sequential read disk I/O throughput of Dom0 and 1, 2, 5, and 10 guest VMs with different block sizes. Figure (c) shows the normalised network throughput of one VM co-resident with another 1 to 9 VMs and the corresponding utilisation of Dom0.

For the disk intensive workload, we run the `fio`[2] benchmark on Dom0 and individual VMs. The application issues requests to the kernel through libraries such as `libc` or `libaio`. On the simple case where the benchmark is configured with an IO Depth of 1. 'fio' will attempt to keep one request in execution at all times, which means as soon as one request completes, 'fio' will send another. We perform sequential reads and vary the block size from 1 KB to 4 MB. Figure 5.2(b) details the total disk throughput of Dom0 and 1 to 10 DomUs processing requests to read a 5 GB file simultaneously. We can split the results into three phases:

(1) For small block sizes (1 KB to 16 KB), we observe a high load (80 to 90%) on Dom0 as it has to process a large number of requests in parallel. In this phase, the total disk throughput is bounded by Dom0's capacity. (2) After that (32 KB to 128 KB) the system is bound by the disk throughput. (3) Once the block size goes beyond 128 KB, the throughput drops for 2, 5, and 10 DomUs while the average utilisation of Dom0 stays at 30 to 40%.

It might be conjectured that the reason for the deterioration in disk throughput with increasing block size may be due to poor I/O subsystem performance, caused by for example, a resource bottleneck of CPU or system bus. However, this is unlikely the case because the green line of 1 guest domain (1 DomU) shows that without contention the guest I/O can perform as well as

[2]Fio benchmark. `http://freecode.com/projects/fio`

Dom0. The grant table contention explains the drop in throughput for the VMs from a block size of 256 KB. The size of a Xen grant table entry is 4 KB [CG05] and the maximum number of entries is limited to 128[3] [PLM+13, CGV07]. As a result, we have: 256 KB/4 KB = 64 table entries and $64 \times 2 = 128$. When the block sizes of more than two VMs are larger than 256 KB, the grant table will be locked when there are too many concurrent disk requests which causes delays and decreases the total disk throughput significantly. We acknowledge that the grant table size is an essential aspect of limiting the application I/O performance. It would be valuable to validate the impact of changing the grant table sizes. However, this involves a lot of efforts on kernel building. In this thesis, we conduct all the I/O experiments with the default size of grant table.

To produce a network intensive workload, we start 10 VMs with `iperf`[4] servers on the same physical machine. We then launch another 10 VMs as clients on other hosts in our local private cloud. All the VMs are connected via 10 Gbps links. Figure 5.2(c) shows the average normalised network throughput and the corresponding average utilisation of Dom0. Throughput is normalised against the performance of when there is only one VM processing network requests. The throughput of the VMs decreases and the mean CPU utilisation of Dom0 increases with a larger number of co-resident VMs. The reason for the drop is a combination of the memory page locking for network requests (see Section 5.2.1) and the scheduling and processing overhead of Dom0.

5.3 System Design

Our benchmarking results demonstrate that VM interference can have a significant impact on performance. We now describe CloudScope, a system that predicts the effects of interference and reacts accordingly to prevent performance degradation. CloudScope runs within each host in the cloud system and is complementary to current resource management components (e.g.

[3]This is set in the Xen kernel via: `MAX_MAPTRACK_TO_GRANTS_RATIO` and `gnttab_max_nr_frames`
[4]Iperf Benchmark. `https://iperf.fr/`

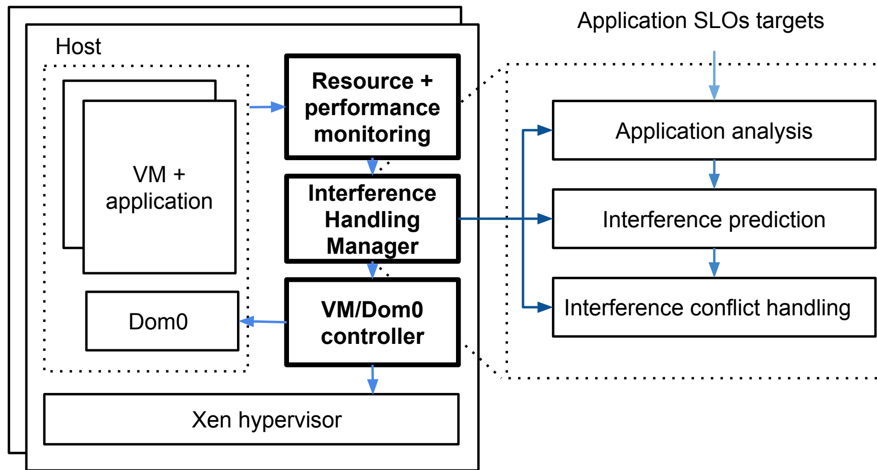


Figure 5.3: CloudScope system architecture

VMware Distributed Resource Scheduler[5] or CloudStack resource manager[6]) that handle SLO violations, dynamic VM scaling and elastic server consolidation [TGS14, NSG+13]. CloudScope incorporates the VM application SLO monitor and evaluator developed in our previous work [CHO+14, CK15]. Figure 5.3 illustrates the overall system architecture which consists of three main parts:

Monitoring Component. The Monitoring Component collects application and VM metrics at runtime. A daemon script reads the resource usage for Dom0 and every VM within Dom0 via `xentop`. The resource metrics include CPU utilisation, memory consumption, disk I/O request statistics, network I/O, number of virtual CPUs (vCPUs), and number of virtual NICs (vNICs). External monitoring tools [CHO+14] are used to keep track of application SLOs in terms of application metrics such as response time, disk/network throughput, or job completion time. The resource and SLO profiling metrics are fed to the Interference Handling Manager.

Interference Handling Manager. The Interference Handling Manager is composed of three main modules. The *application analysis* module analyses the monitoring data from each VM and obtains the application metrics. The result is an initial *application loading vector* for each application VM. The *interference prediction* module incorporates an analytical model based on the *V-slowdown factor* (see Section 5.3.4) that infers the expected application performance

[5]WMware Distributed Resource Scheduler. `https://www.vmware.com/uk/products/vsphere/features/drs-dpm`
[6]CloudStack Resource Manager. `http://cloudstack.apache.org/software/features.html`

degradation from the profile of currently running guest domains and Dom0. The *interference conflict handling* module provides interference-aware scheduling and adaptive Dom0 reconfiguration.

Dom0 Controller. The Dom0 controller calls the corresponding APIs to trigger VM migration or Dom0 reconfiguration based on the prediction results and the SLO targets.

5.3.1 Predicting Performance Interference

Section 5.2 showed that the performance of co-resident CPU, disk, and network intensive applications may decrease due to the paravirtualisation protocol, the load of Dom0, and the number of VMs competing for resources. We can view an application as a sequence of micro job slices accessing different system resources. Each application can be characterised with a certain resource statistic using a *loading vector* [ZT12, LHK+12] that represents the proportion of the time that an application spends on each resource. We define the V-slowdown δ_j of a VM j, as the percentage of degradation in performance due to co-residency compared to no co-residency. We obtain the V-slowdown of an application VM by combining the slowdowns of each resource.

Consider multiple applications running in VMs $1, \ldots, n$ with CPU utilisations $util_1, \ldots, util_n$ on a single PM. A job is considered a sequence of job slices scheduled by the hypervisor to access the physical CPU, memory, disk and network resources. We represent the processing steps of a VM request within a PM as a discrete-time Markov chain in which the states represent the hypervisor layer and physical resources: *Dom0*, *pCPU*, *Disk*, and *Net* as illustrated in Figure 5.4. In this model, we do not deal with memory as a resource as it is by default isolated and efficiently managed in Xen. Phenomenons such as memory ballooning or cache effects [GLKS11] are out of scope for this work.

Each Markov chain in Figure 5.4 represents the processing steps of a specific workload in the Xen virtualised system in Figure 2.3. Note that one job slice can only be processed in one of the four states (*Dom0*, *pCPU*, *Disk*, and *Net*) at any one time. A job moves from state to state based on a probability that depends on the current workloads within the system. In the

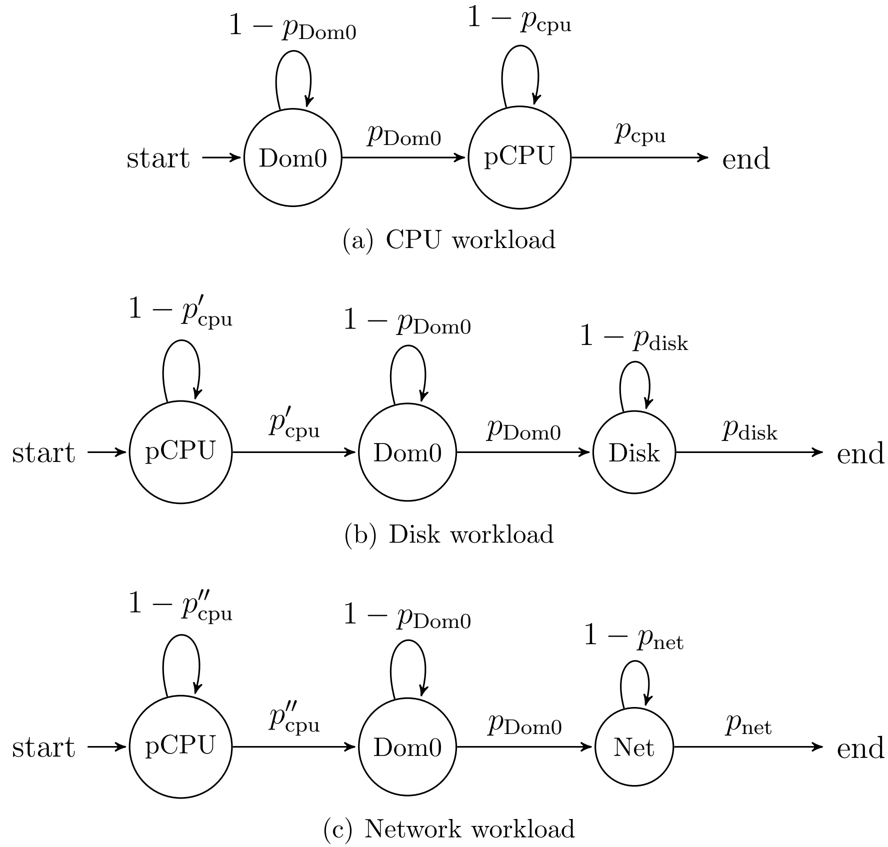


Figure 5.4: State transition diagrams for (a) CPU, (b) disk, and (c) network intensive workloads

following we calculate these probabilities.

5.3.2 CPU workloads

When a CPU job arrives (see Figure 5.4(a)), the CPU scheduler (within hypervisor) has to schedule the vCPU to a runnable queue of a pCPU (physical CPU). p_{Dom0} denotes the probability that in the next interval the job will be forwarded to a pCPU. Assume that the vCPUs of Dom0 are modelled by n M/G/1-PS queues, in which n represents the number of vCPUs and the PS (processor sharing) policy reflects the scheduling policy. In an M/G/1-PS queue, the average time spent in the system by customers that have service time x is denoted by

$$T(x) = \frac{x}{C(1-\rho)}$$

where C is the capacity of the server [HP92] and ρ is the utilisation of the server. As each job sees the same effective service capacity $C(1-\rho)$, the probability of leaving the Dom0 state, i.e., a job is scheduled by Dom0 and assigned to the queue of a pCPU:

$$p_{\text{Dom0}} = 1 - \rho_{Dom0} \tag{5.1}$$

where p_{cpu} represents the probability that a job completes service at a pCPU and leaves the state. The derivation of p_{cpu} is based on the delay caused by oversubscribed resources presented in Section 5.2.

$$p_{\text{cpu}} \approx \begin{cases} 1 & \text{if CPU is not oversubscribed} \\ \frac{\sum_j pCPU_j}{\sum_i util_i \times vCPU_i} & \text{if CPU is oversubscribed} \end{cases} \tag{5.2}$$

where $pCPU_j$ denotes the capacity of the jth physical CPU, $vCPU_i$ denotes the capacity of the ith virtual CPU and $util_i$ denotes the CPU utilisation due to VM_i. If $p_{\text{cpu}} < 1$, then the CPU resources of the physical machine are oversubscribed; otherwise $p_{\text{cpu}} = 1$.

5.3.3 I/O workloads

Recall that the guest VM has a shared memory area with Dom0 for processing I/O requests in Figure 2.3. The Xen hypervisor also has an event channel for handling the interrupts from guest VMs. This channel goes through the hypervisor and has some latency associated with it. Note that we account for this delay in our Dom0 state. When one VM needs to perform I/O, it follows these steps (see also Figures 5.4(b) and 5.4(c)):

1. The VM creates an I/O request and places it in the shared memory area (grant table). This process is represented by state $pCPU$.

2. The VM sends an interruption to Dom0 (state $Dom0$) via a pre-established channel. Dom0 reads the shared memory and asks the hypervisor for access to the memory areas pointed to by this request.

3. Dom0 submits the request either to storage or the network (see state *Disk* and *Net*). When the request completes, Dom0 places the response back in the shared memory, revokes its access to the memory areas pointed to by the request and sends a notification to the guest VM. The guest VM reads the response from the shared memory, clears the interruption channel and accepts the response as success or failure for that request.

Equation 5.3 abstracts the effect of the memory map locking delay, where $\sum_i bs_i$ represents the total I/O sizes of all requests issued at the same time. When this number is larger than the maximum 128 entries $\times$ 4 KB, the memory page locks and updates itself; thus some of the requests have to be placed in the next interval. p'_{cpu} and p''_{cpu} represent the probability of a request successfully accessing the memory table and passing the request to Dom0 for disk and network requests respectively. They depend on p_{cpu} because performing I/O operations also consumes CPU cycles.

$$p'_{\text{cpu}} = p''_{\text{cpu}} \approx \begin{cases} 1 & \text{if } \lfloor \frac{\sum_i bs_i}{128\times 4} \rfloor < 1 \\ \frac{1}{\lfloor \frac{\sum_i bs_i}{128\times 4} \rfloor + 1} \times p_{\text{cpu}} & \text{if } \lfloor \frac{\sum_i bs_i}{128\times 4} \rfloor \geq 1 \end{cases} \tag{5.3}$$

Note that calculation of p'_{cpu} and p''_{cpu} differ slightly as $\sum_i bs_i$ depends on whether it relates to disk or network I/O. In the case of disk I/O, the total I/O size counts both read and write requests. However, we have to count the total I/O size of sending and receiving packets for network I/O separately because they use separate memory tables.

Disk requests are served in FIFO order and thus, the arrival queue length at any disk is equal to $\frac{\rho}{1-\rho}$, where ρ is the utilisation of the server. If the queue length is smaller than 1, meaning there are no queued requests, then the probability of a job accessing the physical disk is $p_{\text{disk}} = 1$. Based on this, the probability of completing disk service is:

$$p_{\text{disk}} = \frac{1}{\frac{\rho_{\text{disk}}}{1-\rho_{\text{disk}}} + 1} = 1 - \rho_{\text{disk}} \tag{5.4}$$

where ρ_{disk} represents the utilisation of the disk channel. For block devices such as iSCSI, which are common in cloud environments, ρ_{disk} would be the utilisation of the connection between

the host and the iSCSI server. In our setup, we have 10 Gbps links between the server and the storage server which comprises multiple disk volumes. We found that in this case p_{disk} is usually close to 1.

The probability of network requests being served and leaving the system can be calculated as,

$$p_{\text{net}} \approx \begin{cases} 1 & \text{if pNIC is not oversubscribed} \\ \frac{\sum_j pNIC_j}{\sum_i util_i \times vNIC_i} & \text{if pNIC is oversubscribed} \end{cases} \tag{5.5}$$

where $pNIC_j$ denotes the capacity of the jth physical network interface, while $vNIC_i$ denotes the capacities of the ith virtual network interface. $util_i$ denotes the network utilisation due to VM_i. Note that we can easily obtain all these parameters, such as physical or virtual CPU utilisation or network capacity, from hypervisor profiling utilities such as `xentop`.

5.3.4 Virtualisation Slowdown Factor

The states of the Markov chains of Figure 5.4 represent a system of inter-related geometric distributions in sequence. Thus the mean time to absorption, i.e. the mean delay for each chain is:

$$\begin{aligned} E(K_{\text{cpu}}) &= \frac{1}{p_{\text{Dom0}}} + \frac{1}{p_{\text{cpu}}} \\ E(K_{\text{disk}}) &= \frac{1}{p_{\text{Dom0}}} + \frac{1}{p'_{\text{cpu}}} + \frac{1}{p_{\text{disk}}} \\ E(K_{\text{net}}) &= \frac{1}{p_{\text{Dom0}}} + \frac{1}{p''_{\text{cpu}}} + \frac{1}{p_{\text{net}}} \end{aligned}$$

$E(K_{\text{cpu}})$, $E(K_{\text{disk}})$ and $E(K_{\text{net}})$ represent the mean time that a VM request will take to complete execution on the CPU, disk or network given a certain workload on the virtualised system. We define $E'(K_{\text{cpu}})$, $E'(K_{\text{disk}})$ and $E'(K_{\text{net}})$ as the expected execution time for a VM running alone or running with other VMs in an environment with unsaturated resources, i.e. when p_{Dom0}, p_{cpu}, p'_{cpu}, p''_{cpu}, p_{disk}, and p_{net} are equal to 1.

Thus, the virtualisation slowdown for each resource given a current workload on the system is:

$$\gamma_{\text{cpu}} = \frac{E(K_{\text{cpu}})}{E'(K_{\text{cpu}})}$$
$$\gamma_{\text{disk}} = \frac{E(K_{\text{disk}})}{E'(K_{\text{disk}})}$$
$$\gamma_{\text{net}} = \frac{E(K_{\text{net}})}{E'(K_{\text{net}})}$$

An application needs a certain proportion of CPU, disk, and network resources to run a job. For example, a file compression job might spend 58% of the total execution time on CPU and 42% on disk I/O. Without any other system bottleneck or competing job, the vector $\beta_{i,j}$ represents an application's resource usage profile, referred to as the loading vector [ZT12, LHK+12, GLKS11].

$$\beta_{i,j} = \frac{\text{the time of job } j \text{ spent on resource } i}{\text{the total completion time}} \tag{5.6}$$

Therefore, the virtualisation slowdown δ of an application/VM when co-located with other VMs on a system with a known current workload is:

$$\delta_j = \sum_i \gamma_i \times \beta_{i,j} \tag{5.7}$$

where j denotes a particular application VM and i represents different types of resources. δ_j allows us to evaluate how much performance slowdown one application VM might experience if co-resident with $n-1$ VMs.

5.4 Interference Conflict Handing

CloudScope can answer several key questions that arise when trying to improve VM scheduling and consolidation in cloud environments. For example: (1) among multiple VM placements which physical machine can best satisfy the required SLO; (2) what should be the right degree of VM consolidation in terms of the utilisation-performance trade-off; (3) can the hypervisor be self-adaptive without having to reboot to improve the performance of applications? In this

section, we illustrate how CloudScope is able to provide insight for answering these questions.

5.4.1 Dynamic Interference Scheduling

Workload consolidation increases server utilisation and reduces overall energy consumption but might result in undesirable performance degradation. By default, all newly created VMs are assigned to a PM by a load balancing scheduler that is generally based on a heuristic such as bin packing.

CloudScope currently decides whether to trigger migration by comparing the V-slowdown factor among all potential PMs, and migrates VMs to the PM with the smallest V-slowdown factor as shown in Algorithm 1. The algorithm is executed when a new virtual machine needs to be launched. In addition, every 60 seconds, CloudScope makes a decision on whether it is necessary to migrate VMs to PMs with less interference or not. The algorithm greedily finds the most suitable PM for each VM by picking the PM with the smallest slowdown when assigned the new VM. It requires the loading vectors from each VM as input. Previous work has shown how to obtain these [ZT12, LHK+12, GLKS11]. In our experiments (see Section 5.5.4) we acquire the loading vectors online from running monitoring tools (such as `top`) inside each VM. This allows us to continuously update and refine migration and consolidation decisions without prior knowledge of the applications.

The time complexity of Algorithm 1 is the product of the number of targeted VMs and PMs, $O(mn)$. However, the V-slowdown model runs simultaneously across the Dom0 of each PM, so in practice the time complexity is linear in the number of targeted VMs.

Algorithm 1 Interference-aware Scheduling Algorithm

Data: Targeted VM_j, where $j \in 1, \ldots, n$;
Resource pool consist of PM_k, where $k \in 1, \ldots, m$;
Obtain the workload factor $\beta_{i,j}$ for each task within VM_j;
Model is the V-slowdown interference prediction model.
Result: VM_j to PM_k assignments
1: **for** $j = 1$ *to* n **do**
2: **for** $k = 1$ *to* m **do**
3: $\delta_j = Predict(\beta_{i,j},\ PM_k,\ Model)$;
4: **end for**
5: **end for**
6: $PM_{candidate} = \min_j(\delta_j)$;
7: $Assign(VM_j, PM_{candidate})$;

5.4.2 Local Interference Handling

In some cases, CloudScope will not migrate the application VM but instead resolve the problem locally using Dom0 reconfiguration. This prevents the application from experiencing a period of high SLO violations and the destination PM from experiencing increased utilisation in Dom0 due to migration.

CloudScope allows adaptive Dom0 configuration to transfer unused resources for better performance without affecting application SLOs. For example, in Figure 5.2(c), 8 vCPUs are given to Dom0. These are needed in order for Dom0 to sustain the workload and fully utilise the hardware resource for the current guests. The same effect can be achieved by changing the CPU weight and cap of Dom0. For example, a domain with a weight of 512 will get twice as much CPU as a domain with a weight of 256 on a contended host. The cap fixes the maximum amount of CPUs a domain can consume. With different weights and caps in the system, we can modify the models by setting:

$$p_{\text{Dom0}} = (1-\rho) * \frac{w_{Dom0}}{\sum_i \frac{w_i}{n}} \tag{5.8}$$

where w_{Dom0} and w_i represent the weight of Dom0 and each guest VM respectively, and n is the number of guest VMs. We assume that the SLOs are provided to CloudScope by the users or cloud providers. Therefore, when a violation of an SLO is detected and the current host

does not have a full CPU utilisation, which in our model means $p_{cpu} = 1$ and $\sum_j pCPU_j > \sum_i util_i \times vCPU_i$, CloudScope will derive the Dom0 CPU weight needed by Equation 5.8. This will change the attributes of file `/boot/extlinux.conf` in Dom0 triggering a hot reconfiguration without rebooting. In Section 5.5.5, we illustrate the effect of modifying the attributes of Dom0 on-the-fly.

5.5 Evaluation

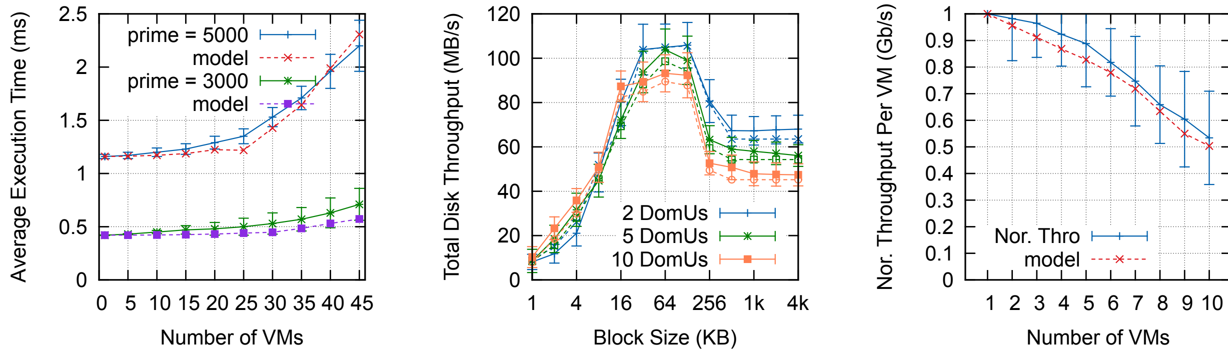
In this section, we evaluate a variety of workloads to validate the proposed model. We use (1) the single workloads for CPU, disk, and network as presented in Section 5.2.2, (2) a synthetic workload in which we combine the three, and (3) a realistic workload which consists of Hadoop MapReduce jobs. The results in this section show that CloudScope is able to accurately capture the performance degradation and the interference of VMs caused by Xen virtualisation. The validation experiments are run with the same hardware configurations as introduced in Table 5.1 of Section 5.2 with the results averaged over 30 runs for each experiment.

5.5.1 Experimental Setup

To ensure that our benchmarking experiments are reproducible, we provide the configuration for the Xen hypervisor used in our experiments. All the experiments are running on XenServer 6.2.0 (Clearwater)[7] with the following configurations: (1) we fine-tune Dom0's vCPU to pCPU affinity before the domains boot. The reason for this is that the vCPU will run on specific NUMA nodes and try to allocate memory closer to it, so it helps Dom0 to deliver better performance; (2) we turn off power saving mode to avoid Xen adjusting the CPU clock rate dynamically[8] (3) we enable hyper-threading on the test machine as the Xen and VMware hypervisors perform better with hyper-threading [CSG13] and to emulate Amazon EC2 in

[7] XenServer – Download. http://xenserver.org/open-source-virtualization-download.html

[8] Tuning XenServer for Maximum Scalability. http://blogs.citrix.com/2012/06/23/xenserver-scalability-performance-tuning/



(a) CPU execution time validation (b) Disk throughput validation (c) Network throughput validation

Figure 5.5: Interference prediction model validation for (a) CPU, (b) disk, and (c) network intensive workloads. Figure (a) shows the validation results for average execution time, executing prime 3000 and 5000. Figure (b) shows the validation results for the total disk I/O throughput of 2, 5, and 10 VMs with all block sizes. Model results are shown in the same color but as dashed lines. Figure (c) shows the validation results of the normalised throughput with measurement standard deviation of one VM co-resident with another 1 to 9 VMs.

which VM instances run on hyper-threaded CPUs. The system used to collect the performance data from our benchmarks is similar to the testbed setup in [CK15].

For disk I/O, before running the actual experiments, we create a large file on each VM to completely fill the file system and then delete this file. This ensures that the virtual hard disk is fully populated and performance metrics will be correct. We also run `iostat` to make sure that all the virtual block devices are active.

5.5.2 CPU, Disk, and Network Intensive Workloads

First we present the prediction results for the scenarios in which the VMs are running CPU, disk, or network intensive workloads generated by `sysbench`, `fio`, and `iperf`, respectively (equal to Section 5.2). We validate our model against system measurements presented in Figure 5.5. Figure 5.5(a) shows the validation results of the average execution time of prime 3000 and prime 5000 workloads with an increasing number of VMs. The mean model prediction error is 3.8% for prime 5000 and 10.5% for prime 3000. From 1 to 25 VMs, the model deviates from the measurement data but the results are still within standard deviation of the measurements. In particular, the model underestimates execution time for the prime 3000 workload, as the

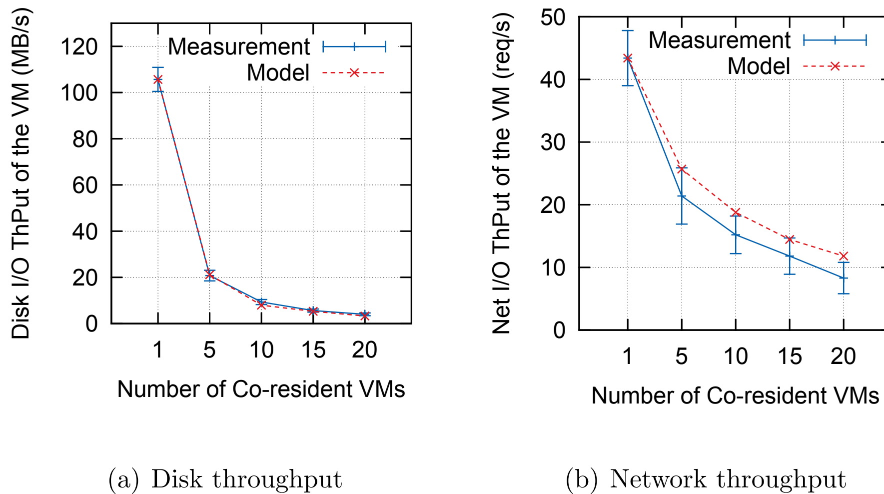
performance is affected by hyper-threading overhead which is not considered in the model.

The disk intensive workload validation results are shown in Figure 5.5(b). The dashed lines represent the model predictions. The total throughput of 2, 5, and 10 VMs running sequential read workloads with varying block sizes from 1 to 4 MB is predicted with a mean error of 8.6%. Three stages as we discussed in Section 5.2.2 including the drops at around 256 KB block size in all three scenarios are precisely captured. As shown in Figure 5.2(b), block sizes from 1 to 32 KB are dominated by the hypervisor overhead caused by the performance differences between non-virtualised and virtualised domains. Secondly, block sizes from 32 KB to 256 KB, the system is bound by the disk throughput. Thirdly, for block sizes larger than 512 KB, our model can account for the processing of memory page locking and updating by capturing the fact that the disk throughput are limited by the size of the grant table.

Figure 5.5(c) represents the normalised network throughput validation with 1 to 9 VMs running `iperf`. The vertical bars represent the standard deviation of the throughput measurements. The mean prediction error is 4.8%. This shows that our model follows the measurements closely and can reflect the effect of intensive Dom0 overhead and sharing network resources with other co-resident VMs.

5.5.3 Mixed Workload

Next, we apply our model to a workload consisting of a mix of disk I/O and network intensive jobs in combination with a moderate CPU workload running together within each VM. The VMs were configured with 4 vCPUs with 1 GHz per vCPU, 8 GB memory, 32 GB storage, and 1 vNIC. We refer to the network intensive workload as *std-net*. It comprises a set of HTTP requests over the network to the VM which runs an HTTP server. `httperf` was deployed to generate the HTTP client workload with clients distributed across the hosts in our private departmental cloud. The disk intensive workload is referred to as *std-disk* which is a sequential read of 1 GB data with block size 64 KB and without buffer access. The CPU workload is referred to as *std-cpu* which is a `sysbench` prime 3000 workload.



(a) Disk throughput (b) Network throughput

Figure 5.6: Interference prediction model validation of mixed workload. Figure (a) and (b) show the disk and network throughput of a targeted VM collocated with 5, 10, 15, and 20 VMs

We predict the virtualisation slowdown of a VM running a mixed workload when co-resident with 5, 10, 15, and 20 VMs running the same mixed workloads. Figure 5.6 shows the prediction of the model for the disk and network throughput for a target VM as the number of co-resident VMs increases. The *std-cpu* workload (not shown) did not incur obvious degradation resulting from co-residency (also predicted by the model). The average relative prediction error is less than 8.3% and 18.4% for disk and network throughput respectively. The prediction error of network throughput is higher than disk throughput likely due to other cloud network traffic between our client and server VMs.

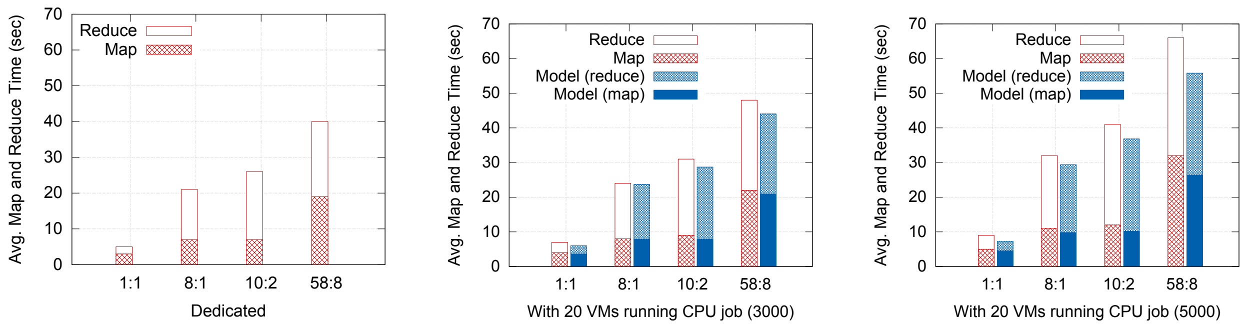
5.5.4 MapReduce Workload

In this section we investigate the accuracy of our interference prediction model based on a real workload. We deploy Apache Hadoop Yarn[9] on 4 VMs (4 2.9 GHz vCPUs, 8 GB memory, and 40 GB local disk) co-located within the same physical machine with one master node and three workers. We use Apache Pig[10] to run a subset of the queries from the TPC-H benchmark[11] on a 10 GB dataset with varying complexities. Some of the queries need several consecutive

[9]Apache Hadoop Yarn. `http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html`

[10]Apache Pig. `http://pig.apache.org/`

[11]TPC-H benchmark. `http://www.tpc.org/tpch//`



(a) 4 MapReduce jobs on dedicated PM (b) Map and Reduce time prediction with 20 VMs executing prime 3000 (c) Map and Reduce time prediction with 20 VMs executing prime 5000

Figure 5.7: Interference prediction model validation for different Hadoop workloads with different numbers of mappers and reducers. For example 1:1 represents 1 mapper and 1 reducer. Figure (a) shows each Hadoop Yarn job running dedicated within one PM. Figure (b) and (c) show the validation results of each Hadoop job with 20 co-resident VMs executing prime 3000 and 5000, respectively.

MapReduce jobs to be processed. We run the first four TPC-H queries for 30 times both alone and with another 20 VMs running prime 3000 and 5000. For space reasons, we do not list all the MapReduce jobs involved during a query but rather pick individual jobs from each query in a way that we cover a broad spectrum of different numbers of map and reduce tasks. Each job has M mappers and R reducers, written as $M{:}R$.

Figure 5.7 presents the average performance of 1:1, 8:1, 10:2, and 58:8 jobs running alone and co-located with the other 20 VMs running either prime 3000 or 5000. The model predictions are compared to measurements. Map and reduce tasks were validated separately because they have different proportions of resource usage. The mean relative error for map and reduce tasks is 10.4% and 11.9%, respectively. The mean relative errors for each 1:1, 8:1, 10:2 and 58:8 job are 14.3%, 6.7%, 8.3% and 14.8%.

The evaluation shows that in this case study, our model is able to achieve a prediction error less than 20% across all workloads. In the following, we use our model to enhance VM performance by implementing an interference-aware scheduling approach and an adaptive Dom0.

PMs	13 Dell PowerEdge C6220 & 8 Dell PowerEdge C6220-II
VMs	250 to 270 other VMs running during the experiments
Shared Storage	NetApp + Cumulus 04 (ZFS) shared storage (iSCSI)
Network	10Gbps Ethernet. 10Gbps switch
IaaS	Apache Cloudstack v4.1.1
Hypervisor	XenServer 6.2.0

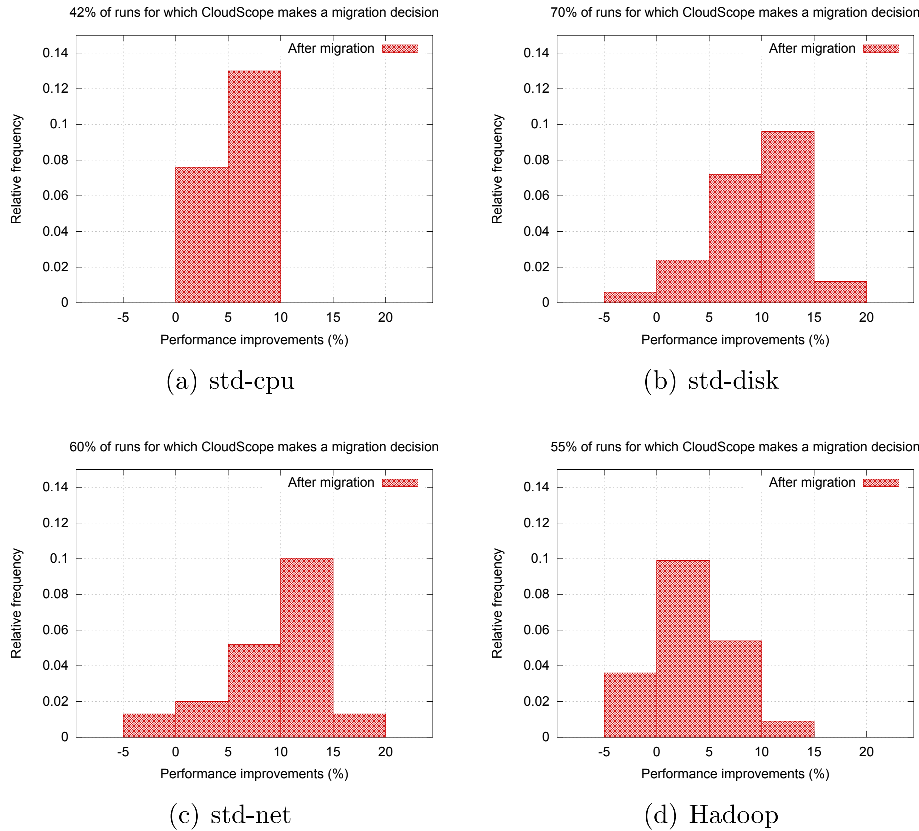
Table 5.2: Specifications for interference-aware scheduler experimental environment

5.5.5 Interference-aware Scheduling

To evaluate CloudScope's interference-aware scheduling mechanism, we compare it to the default CloudStack VM scheduler. Using the experimental setup described in Table 5.2, we utilise our private cloud consisting of 21 physical machines running Apache CloudStack. At the time of our experiments, the cloud was open to other users and was running between 250 to 270 other VMs with unknown workloads. We prepare the 34 VMs with the following workloads: 10 VMs running *std-cpu*, 10 VMs running *std-disk*, and 10 VMs running *std-net* and another 4 VMs running Hadoop as configured in the previous section.

We launched these 34 VMs (including 10 *std-cpu*, 10 *std-disk*, 10 *std-net* and 4 Hadoop VMs) using both the default CloudStack VM scheduler and our CloudScope scheduler and measured the average execution time, throughput, and Map/Reduce job completion times for all VMs. We repeated the *std-cpu*, *std-disk* and *std-net* experiments 5 times and we conduct the Map/Reduce experiments 10 times. Since we conduct this experiment in a real clouds environment, we do not have access to all the applications running in the cloud. We assume that the I/O requests of other tenants that are larger than 10 IOPS (Input/Output Operations per Second) are small random read/write with 8KB block size; the rest of the I/O requests are sequential read/write with 64KB block size. CloudScope does not make a migration decision unless the performance gain on the new PM destination is over 5%, so what we expect to see is that for some cases CloudScope does not migrate the VM instances. Also, due to the dynamic nature of the Cloud environment, it is not guaranteed that CloudScope can always make the ideal decision with performance improvement after VM migration or scheduling.

Figure 5.8 details the performance improvement histograms of CloudScope over CloudStack



(a) std-cpu (b) std-disk

(c) std-net (d) Hadoop

Figure 5.8: Histograms of the performance improvement of the CloudScope interference-aware scheduler over the CloudStack scheduler, for those cases where CloudScope migrates some of the VMs to the PM with the smallest interference. Figures (a), (b), (c) and (d) show the results of CPU, disk and network intensive jobs, and Hadoop job respectively.

scheduler of different jobs, in which the x-axis shows different ranges of the percentile performance improvements. We compare the job completion time results under CloudScope to the ones under CloudStack scheduler. The title of each plot shows the percentage of runs CloudScope decided to make a migration decision, which means that in some cases CloudScope scheduler accepted the VM placement decision made by CloudStack. As illustrated in Figure 5.8(a), migration decision was made for only 42% of the total runs of cpu-intensive jobs, while migration decision was made for 70% and 60% of the total runs of disk and network-intensive jobs respectively. This is due to CPU resource is more sufficient in the cloud compared to disk and network resources. For some cases in Figure 5.8(b) and Figure 5.8(c), CloudScope can obtain more than 15% performance improvement for both disk and network-intensive jobs maximumly; however, a few cases in the range of $[-5, 0]$ show that the migration decisions do not always guarantee the overall performance without any knowledge of the future workload

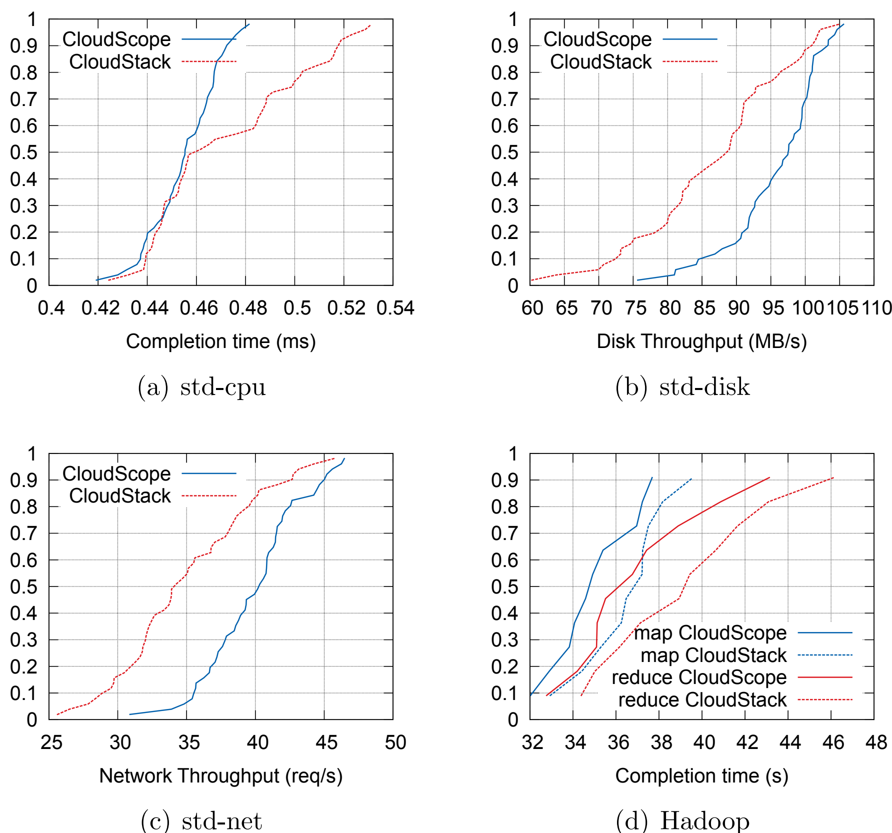


Figure 5.9: CDF plots for the job completion times of different tasks under CloudScope compared to CloudStack, for those cases where CloudScope migrates some of the VMs to the PM with the smallest interference. Figures (a), (b), (c) and (d) show the results of CPU, disk and network intensive jobs, and Hadoop job respectively.

and the dynamics.

Figure 5.9 shows the CDF (Cumulative Distribution Function) of the job completion times of CPU, disk, network-intensive and Map/Reduce job under CloudScope compared to CloudStack. This provides an overview of the percentile distribution of the completion times of different jobs. Figure 5.9(b) shows that CloudScope does not make migration decision for 60% of the total numbers of runs. Also, it shows that with CloudScope, std-cpu completion time can be greatly improved by reducing the 90th percentile of completion time from 0.53 ms to 0.47 ms. The job completion time under CloudScope is between 0.42 to 0.46 ms. In addition, Figure 5.9(b) and Figure 5.9(c) both show that CloudScope manages to migrate VMs to a PM with smaller disk or network I/O interference to avoid the presence of the long tail throughput. For example, Figure 5.9(b) presents that CloudScope significantly improve the 10th percentile of disk throughput from 60 MB/s to 75 MB/s. Last, Figure 5.9(c) illustrates the completion

time of the task with 58 mappers and 8 reducers. It shows CloudScope can improve both mapper and reducer completion time.

Figure 5.10 shows the average performance improvement of each type of VM when scheduled with CloudScope in comparison to the default CloudStack scheduler. The error bars show the standard deviation across all VMs for a single workload and all runs. The *std-disk* and *std-net* VMs show a performance improvement of 10% when scheduled with CloudScope. The *std-cpu* and Hadoop VMs show an improvement of 5.6% and 2.1%, respectively. The *std-cpu* VMs do not obtain significant improvement, this is due to the CPU resource of our cloud platform is not oversubscribed. Because of the grant memory management (see Section 5.2.2), the I/O intensive VMs are more sensitive to resource contention and hence, the CloudScope scheduler achieves larger improvements in these cases. However, because we only migrate the VM image but without the data image for Hadoop. Due to the data locality problem, the Hadoop workload might experience high network communication instead of being benefit from lower performance interference. Also, the estimation of other co-located applications also limits the performance improvement.

5.5.6 Adaptive Control Domain

We also use CloudScope to implement an adaptive Dom0 which is able to its configuration parameters at runtime. We run the *std-net* workload using VM-net VMs (see Table 5.1) using the same PM. 10 VMs running `iperf` server were launched on the PM. Dom0 was configured with 4 vCPUs with weight 256 and a cap of 400. CloudScope could recognise that Dom0 suffered heavy network I/O workload while handling the network traffic. CloudScope obtained the weight needed by Equation 5.7 and 5.8, calculating a new weight of > 425. Changing the weight of Dom0 to 512 provides an average performance gain of 28.8% (see Figure 5.11).

The two presented simple use cases demonstrate that our model can be successfully applied to improve VM performance. We believe that more sophisticated scheduling and reconfiguration approaches can also benefit from CloudScope.

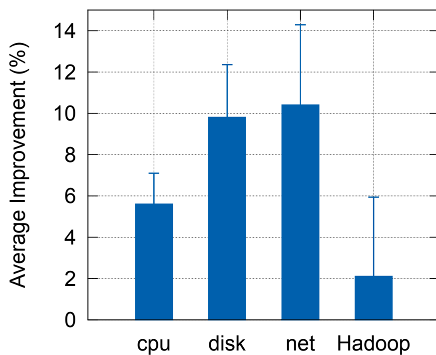


Figure 5.10: CloudScope scheduling results compared to the default CloudStack scheduler

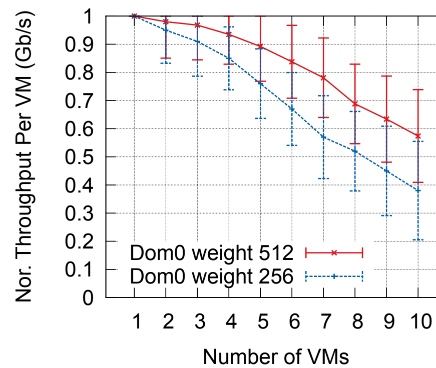


Figure 5.11: CloudScope self-adaptive Dom0 with different vCPU weights

Approaches	Application	Techniques	Avg. Err.
TRACON [CH11]	Data-intensive	Model training & Nonlinear	19%
CloudScale [SSGW11]	CPU contention	Linear regression	14%
CPI^2 [ZTH$^+$13]	CPU contention	Correlation function	n/a
Paragon [DK13]	CPU or IO-bound	Microbenchmark & ML	5.3%
Cuanta [GLKS11]	Cache, memory	Cache pressure clone	4%
Q-Clouds [NKG10]	CPU contention	Nonlinear	13%
CloudScope [CRO$^+$15]	Any application	No microbenchmark & linear model	9%

Table 5.3: Related work of virtualisation interference prediction (ML represents machine learning)

5.6 Summary

CloudScope is lightweight and provides good prediction accuracy in comparison to previous work. Table 5.3 compares CloudScope to similar work in the literature. The main difference is that, CloudScope works for any application with the smallest prediction error and no pre-training or microbenchmark is needed for the prediction.

In this chapter, we have presented CloudScope, a comprehensive system to predict resource interference in virtualised environments, and used the prediction to optimise the operation of a cloud environment. CloudScope incorporates a lightweight analytical model for the prediction of performance degradation. To this end, we have characterised and analysed different workload behaviours by benchmarking CPU, disk, and network intensive workloads. CloudScope predicts the completion time of jobs by a using virtualisation-slowdown factor that incorporates the effect of hypervisor overhead and resource contention. Evaluation shows that CloudScope can

achieve an average relative error of 6.1% for single resource intensive workloads, 13.4% for mixed resource workloads, and 11.2% for MapReduce workloads. CloudScope provides an efficient interference-aware VM scheduling mechanism which can improve job completion time on average by 7.5%.

Chapter 6

Conclusion

This dissertation has addressed the question of how to effectively diagnose performance problems and manage application performance in large-scale virtualised multi-tenant cloud environments. We have shown that a lightweight analytical model based on comprehensive monitoring data can in many cases provide an efficient solution.

Many related methods and systems have attended to solve different aspects of this problem. Our work fills some gaps in the areas of monitoring, modelling and managing and combines them in a coherent framework. Specifically, we begin by showing how a cloud benchmarking system or a performance management system can benefit from graphical and intuitive specification of performance queries and SLOs using Performance Trees. We next show that a model can tell us how many VMs of different granularities (e.g. different numbers of CPU cores) should serve a workload in light of the budget and performance trade-off. Finally, we show how a lightweight and general performance interference prediction model can identify bottlenecks and enable intelligent interference-aware resource scheduling policies. In an era of cloud-based systems that are rapidly evolving, we believe that simple, modular-based frameworks for performance management can enable rapid innovation. Indeed, our Performance Tree-based automatic control platform is currently only 9 000 lines of code, and our frameworks for multicore scalability and performance interference prediction are significantly smaller.

Even though we readily acknowledge that our methods and techniques will not solve *all*

performance-related problems in this area, we believe that our *monitoring*, *modelling* and *managing* methodology provides a useful reference approach for performance engineering in clouds. In the rest of this chapter, we summarise our most important contributions and outline some possibilities for future work.

6.1 Summary of Achievements

Performance monitoring and performance evaluation are fundamental to track the health of cloud applications and to ensure SLA compliance. To enable this, performance engineers should have the ability to specify complex SLOs in an intuitive accessible way. To this end, in chapter 3 we present a Performance Tree-based monitoring and resource control platform for applications running in clouds which makes the following contributions:

- We conduct extensive benchmarking experiments and present some lessons we learned related to accurate and reproducible monitoring in a virtualised environment.
- We discuss system requirements for an extensible modular system which allows for monitoring, performance evaluation and automatic scaling up/down control of cloud-based applications.
- We present a front-end which allows for the graphical specification of SLOs using PTs. SLOs may be specified by both live and historical data, and may be sourced from multiple applications running on multiple clouds.
- We demonstrate how our monitoring and evaluation feedback loop can ensure that the SLOs of a web application are met by auto-scaling.

To leverage monitoring data efficiently, cloud managers and end users are keen to understand the relationship between their applications and the system resources in multicore virtualised environments. In Chapter 4 we try to understand the scalability behaviour of network/CPU intensive applications running on a virtualised multicore architecture. This work makes the following contributions:

- We develop a multi-class queuing model for predicting the performance of network-intensive applications deployed in multicore virtualised environments.

- We derive an approximate analytical solution and validate our model in our testbed. The validation results show that the model is able to predict the expected performance in terms of response time, throughput and CPU utilisation with relative errors ranging between 8 and 26%

- We present a means to achieve better scalability for multicore web servers through the use of virtual hardware. We also demonstrate the applicability of our model in predicting the performance of different system configurations.

In the corresponding validation experiments, we found that the *service rate* of the vCPU core varies with different set of co-resident applications. As a result, the idea of the performance interference prediction and management came to us. Thus, instead of requiring comprehensive micro-benchmarks, online training or complex parametrisation, we seek flexible lightweight models that can be deployed in cloud environments where applications and workloads change frequently. We make the following contributions:

- We introduce a lightweight analytical model solely based on available system parameters which predicts the impact of co-residing VMs on performance.

- Evaluation shows that CloudScope (the system we implement) interference prediction can achieve an average relative error of 6.1% for single resource intensive workloads, 13.4% for mixed resource workloads, and 11.2% for MapReduce workloads with respect to real system measurements.

- CloudScope provides an efficient interference-aware VM scheduling mechanism which can improve job completion time on average by 7.5% compared to the default scheduler and the adaptive control domain can provide an average performance gain of 28.8% with respect to the default configuration.

6.2 Future Work

There are several avenues to be explored in future work as follows:

- **New model to solve the limitations:** We outlined some limitations of our multicore performance prediction model in Section 4.4.4. We identified limitations with respect to routing probability, interrupt priority, context switching overhead and hypervisor overhead. Consideration of hypervisor overhead has already been accounted for in our subsequent development of CloudScope. For other limitations, possible directions would be to use a modelling technique such as multi-class processor sharing queues with priority.

- **Performance diagnosing:** The Performance Tree-based platform introduces a flexible way of monitoring and evaluating performance through intuitive graphical performance query specification. However, it is sometimes difficult to have an intuition about what results in good performance and what causes performance bottlenecks. This is especially true when users are dealing with large-scale distributed systems, such as Hadoop Yarn, Spark, Flink etc. Factors such as communication between multiple nodes, memory overhead of various data representations, CPU overhead of data serialisation, data skew can significantly impact the performance. In CloudScope we also introduce the idea of 'diagnosing' where each parameter in the model is associated with a corresponding hardware or hypervisor attribute. With the similar idea and comprehensive profiling, it would be challenging to develop tools that can automatically detect these inefficiencies, and give users directions about the sensitivity of application performance to hardware and hypervisor attributes.

- **Workload-aware VM consolidation:** Although CloudScope interference-aware scheduler supports flexible VM consolidation through minimising the interference slowdown-factor, which has already been implemented, finding the right policies for consolidating VMs in terms of, e.g. different applications, various SLO requirements etc, remains an open challenge. CloudScope is designed to enable complex polices that are easy to change. Some possible directions can be, for example, how can VMs be consolidated and scheduled

to enforce priorities and individual SLOs? How can a workload-aware scheduler reduce the impact of contention in a last-level shared cache, while minimising the contention on the disk and network I/O?

- **Self-adaptive hypervisor:** CloudScope currently allows adaptive Dom0 configuration on-the-fly to improve the job completion time and aggregated throughput, but there are other requirements such as per-VM SLO guarantees that can benefit from this approach. Exploring more algorithms based on this idea could improve application performance and create more opportunities for on-line system optimisation and performance guarantee mechanisms.

- **Multiple control domains:** We currently refer to Dom0 as control domain; however, this is going to change. With the observed evolution of hardware towards NUMA architectures and servers with several dozen cores, Xen is now supporting multiple "driver domains"[1]. This means we wil have separate domains for handling network and storage I/O. Multiple control domains is the future development of virtualisation technology. CloudScope models CPU, disk and network requests individually, which gives a better way of understanding how each type of request consumes each resource. Thus, it would be interesting to have a mechanism which can decide the size of each mini-domain to serve each of the corresponding guest domains.

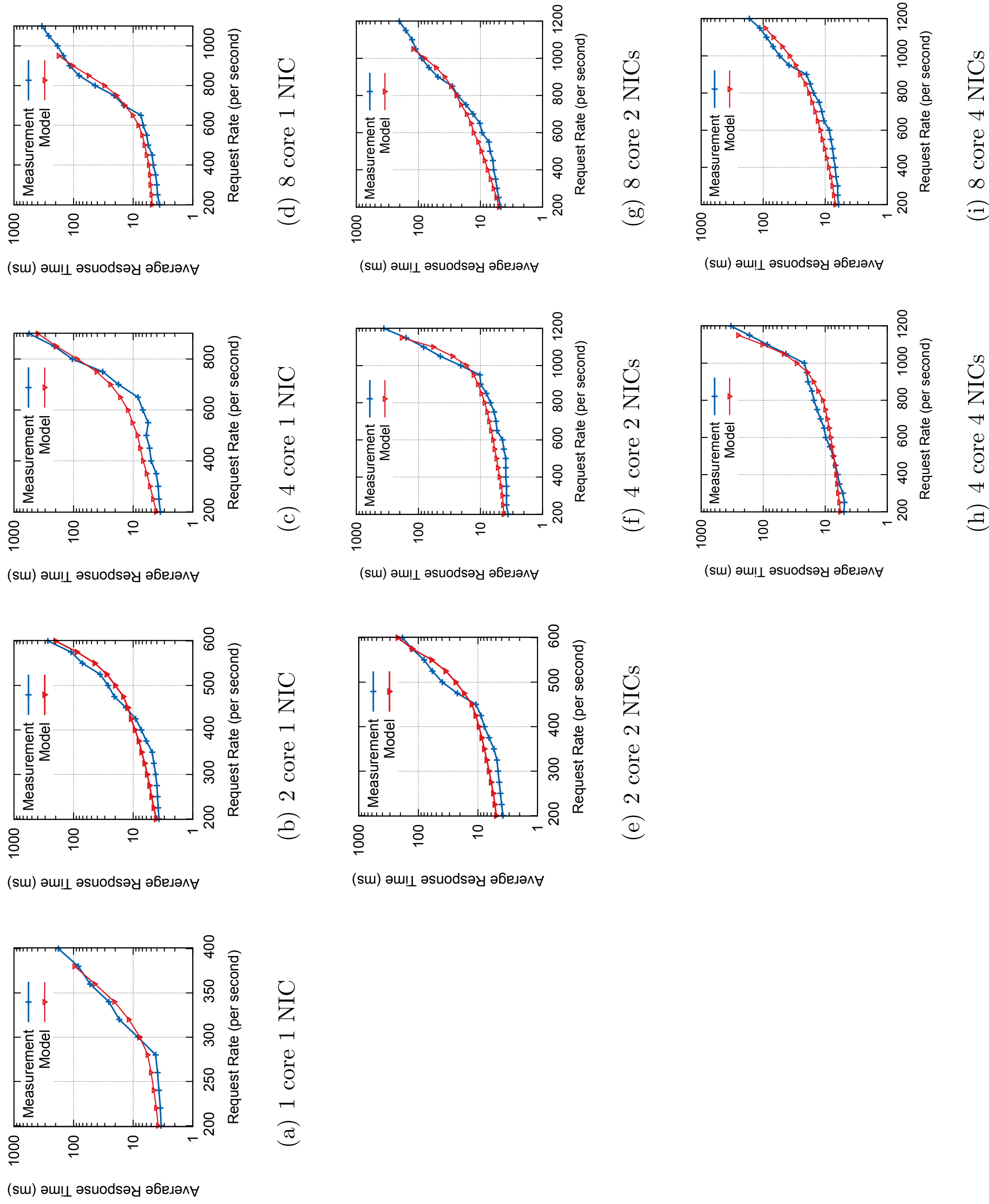
As cloud environments and applications grow more complex and users start demanding more reliable services, these policies will be important to maintain good performance. We hope that continued experience and research with the performance engineering in clouds will help address these challenges, and lead to solutions that improve the experiences of cloud managers and users.

[1] Device passthrough to driver domain in Xen. `http://wiki.xenproject.org/mediawiki/images/1/17/Device_passthrough_xen.pdf`

Appendix A

Xen Validation Results

The validation results we present in Section 4.4 is based on a Type-2 hypervisor, namely VirtualBox. Here we present the corresponding results of a Type-1 hypervisor, Xen, as shown in Figure A.1. The infrastructure setup is illustrated in Section 4.2 Figure 4.1, using the default configurations. The hypervisor runs on an IBM System X 3750 M4 with four Intel Xeon E5-4650 eight-core processors at 2.70GHz to support multicore VM instances comprising from 1 to 8 cores. Each virtual CPU is configured with 2GHz frequency.



(a) 1 core 1 NIC

(b) 2 core 1 NIC

(c) 4 core 1 NIC

(d) 8 core 1 NIC

(e) 2 core 2 NICs

(f) 4 core 2 NICs

(g) 8 core 2 NICs

(h) 4 core 4 NICs

(i) 8 core 4 NICs

Figure A.1: Validation of response time of 1 to 8 core with multiple number of NICs on Xen hypervisor

Bibliography

[ABG15] George Amvrosiadis, Angela Demke Brown, and Ashvin Goel. Opportunistic storage maintenance. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, pages 457–473. ACM, 2015.

[ABK+14] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. End-to-end performance isolation through virtual datacenters. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 233–248, 2014.

[AFG+10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[ALW15] Marcos K Aguilera, Joshua B Leners, and Michael Walfish. Yesquel: scalable SQL storage for Web applications. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, pages 245–262. ACM, 2015.

[AM13] Arwa Aldhalaan and Daniel A Menascé. Analytic performance modeling and optimization of live VM migration. In *Computer Performance Engineering*, pages 28–42. Springer, 2013.

[ASR+10] Sherif Akoush, Ripduman Sohan, Andrew Rice, Andrew W Moore, and Andy Hopper. Predicting the performance of virtual machine migration. In *Proceedings of the 18th International Symposium on Modeling, Analysis and Simulation of Com-*

puter and Telecommunication Systems (MASCOTS), pages 37–46. IEEE/ACM, 2010.

[BBM+13] Piotr Bar, Rudy Benfredj, Jonathon Marks, Deyan Ulevinov, Bartosz Wozniak, Giuliano Casale, and William J Knottenbelt. Towards a monitoring feedback loop for cloud applications. In *Proceedings of the International Workshop on Multi-cloud Applications and Federated Clouds (MODAClouds)*, pages 43–44. ACM, 2013.

[BCKR11] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards predictable datacenter networks. In *Proceedings of the ACM SIGCOMM Conference*, volume 41, pages 242–253. ACM, 2011.

[BCMP75] Forest Baskett, K Mani Chandy, Richard R Muntz, and Fernando G Palacios. Open, closed, and mixed networks of queues with different classes of customers. *Journal of the ACM (JACM)*, 22(2):248–260, 1975.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP)*, volume 37, pages 164–177. ACM, 2003.

[BDK+08] Darren K Brien, Nicholas J Dingle, William J Knottenbelt, Harini Kulatunga, and Tamas Suto. Performance trees: Implementation and distributed evaluation. In *Proceedings of the 7th International Workshop on Parallel and Distributed Methods in Verification (PDMC)*, 2008.

[BGHK13] Fabian Brosig, Fabian Gorsler, Nikolaus Huber, and Samuel Kounev. Evaluating approaches for performance prediction in virtualized environments. In *Proceedings of the 21st International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 404–408. IEEE, 2013.

[BGL+10] Collin Bennett, Robert L Grossman, David Locke, Jonathan Seidman, and Steve Vejcik. Malstone: towards a benchmark for analytics on large data clouds. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 145–152. ACM, 2010.

[BKKL09] Carsten Binnig, Donald Kossmann, Tim Kraska, and Simon Loesing. How is the weather tomorrow? Towards a benchmark for the cloud. In *Proceedings of the Second International Workshop on Testing Database Systems (DBTest)*, page 9. ACM, 2009.

[BM13] Shouvik Bardhan and Daniel A Menascé. Analytic Models of Applications in Multi-core Computers. In *Proceedings of the 21st International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 318–322. IEEE, 2013.

[BRX13] Xiangping Bu, Jia Rao, and Cheng-zhong Xu. Interference and locality-aware task scheduling for MapReduce applications in virtual clusters. In *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing (HPDC)*, pages 227–238. ACM, 2013.

[BS10] Sean Kenneth Barker and Prashant Shenoy. Empirical evaluation of latency-sensitive application performance in the cloud. In *Proceedings of the 1st Annual ACM SIGMM Conference on Multimedia Systems (MMSys)*, pages 35–46. ACM, 2010.

[BWT12] Salman A Baset, Long Wang, and Chunqiang Tang. Towards an understanding of oversubscription in cloud. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE)*, 2012.

[CANK03] Jianhua Cao, Mikael Andersson, Christian Nyberg, and Maria Kihl. Web server performance modeling using an M/G/1/K* PS queue. In *Proceedings of the 10th*

International Conference on Telecommunications (ICT), volume 2, pages 1501–1506. IEEE, 2003.

[CCD+01] Graham Clark, Tod Courtney, David Daly, Dan Deavours, Salem Derisavi, Jay M Doyle, William H Sanders, and Patrick Webster. The Möbius modeling tool. In *Proceedings of the 9th International Workshop on Petri Nets and Performance Models (PNPM)*, pages 241–250. IEEE, 2001.

[CCS+15] Oliver RA Chick, Lucian Carata, James Snee, Nikilesh Balakrishnan, and Ripduman Sohan. Shadow Kernels: A General Mechanism For Kernel Specialization in Existing Operating Systems. In *Proceedings of the 6th Asia-Pacific Workshop on Systems (APSys)*, pages 1–7. ACM, 2015.

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4–16, 2008.

[CDM+12] Jianjun Chen, Chris Douglas, Michi Mutsuzaki, Patrick Quaid, Raghu Ramakrishnan, Sriram Rao, and Russell Sears. Walnut: a unified cloud object store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 743–754. ACM, 2012.

[CFF14] Antonio Corradi, Mario Fanelli, and Luca Foschini. VM consolidation: A real case based on OpenStack Cloud. *Future Generation Computer Systems*, 32:118–127, 2014.

[CG05] Ludmila Cherkasova and Rob Gardner. Measuring CPU Overhead for I/O Processing in the Xen Virtual Machine Monitor. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, volume 50, 2005.

[CGPS13] Davide Cerotti, Marco Gribaudo, Pietro Piazzolla, and Giuseppe Serazzi. End-to-End performance of multi-core systems in cloud environments. *Computer Performance Engineering*, pages 221–235, 2013.

[CGV07] Ludmila Cherkasova, Diwaker Gupta, and Amin Vahdat. Comparison of the three CPU schedulers in Xen. *ACM SIGMETRICS Performance Evaluation Review*, 35(2):42–51, 2007.

[CH11] Ron C Chiang and H Howie Huang. TRACON: interference-aware scheduling for data-intensive applications in virtualized environments. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, page 47. ACM, 2011.

[Che07] Sing-Kong Cheung. *Processor sharing queue and resource sharing in wireless LANs*. PhD thesis, University of Twente, 2007.

[CHHW14] Ron C Chiang, Jinho Hwang, H Howie Huang, and Timothy Wood. Matrix: Achieving predictable virtual machine performance in the clouds. In *11th International Conference on Autonomic Computing (ICAC)*, pages 45–56, 2014.

[CHO+14] Xi Chen, Chin Pang Ho, Rasha Osman, Peter G Harrison, and William J Knottenbelt. Understanding, modelling, and improving the performance of web applications in multicore virtualised environments. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 197–207. ACM, 2014.

[CK15] Xi Chen and William Knottenbelt. A Performance Tree-based Monitoring Platform for Clouds. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 97–98. ACM, 2015.

[CKK11] Giuliano Casale, Stephan Kraft, and Diwakar Krishnamurthy. A model of storage I/O performance interference in virtualized systems. In *Proceedings of the 31st International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 34–39. IEEE, 2011.

[CMF+14] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet

services. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 217–231, 2014.

[Col16] Louis Columbus. Roundup Of Cloud Computing Forecasts And Market Estimates, 2015. Forbes/Tech. `http://docs.aws.amazon.com/general/latest/gr/aws-ip-ranges.html`. Accessed 14 June 2016.

[CRO+15] Xi Chen, Lukas Rupprecht, Rasha Osman, Peter Pietzuch, Felipe Franciosi, and William Knottenbelt. CloudScope: Diagnosing and Managing Performance Interference in Multi-Tenant Clouds. In *Proceedings of the 23rd International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 164–173. IEEE, 2015.

[CRS+08] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. In *Proceedings of the 34th International Conference on Very Large Data Bases (VLDB)*, volume 1, pages 1277–1288. VLDB Endowment, 2008.

[CSA+14] Lydia Y Chen, Giuseppe Serazzi, Danilo Ansaloni, Evgenia Smirni, and Walter Binder. What to expect when you are consolidating: effective prediction models of application performance on multicores. *Cluster computing*, 17(1):19–37, 2014.

[CSG13] Faruk Caglar, Shashank Shekhar, and Aniruddha Gokhale. A Performance Interference-aware Virtual Machine Placement Strategy for Supporting Soft Real-time Applications in the Cloud. *Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN, USA, Tech. Rep. ISIS-13-105*, 2013.

[CST+10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, pages 143–154. ACM, 2010.

[CWM+14] Brendan Cully, Jake Wires, Dutch Meyer, Kevin Jamieson, Keir Fraser, Tim Deegan, Daniel Stodden, Geoffre Lefebvre, Daniel Ferstay, and Andrew Warfield.

Strata: High-performance scalable storage on virtualized non-volatile memory. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST)*, pages 17–31, 2014.

[DHK04] Nicholas J Dingle, Peter G Harrison, and William J Knottenbelt. Uniformization and hypergraph partitioning for the distributed computation of response time densities in very large Markov models. *Journal of Parallel and Distributed Computing*, 64(8):908–920, 2004.

[DK13] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, volume 41, pages 77–88. ACM, 2013.

[DKS09] Nicholas J Dingle, William J Knottenbelt, and Tamas Suto. PIPE2: a tool for the performance evaluation of generalised stochastic Petri Nets. *ACM SIGMETRICS Performance Evaluation Review*, 36(4):34–39, 2009.

[DP11] Jeremiah D Deng and Martin K Purvis. Multi-core application performance optimization using a constrained tandem queueing model. *Journal of Network and Computer Applications*, 34(6):1990–1996, 2011.

[EBA+11] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376. IEEE, 2011.

[Erl09] Agner Krarup Erlang. Sandsynlighedsregning og telefonsamtaler (In Danish, translated: The theory of probabilities and telephone conversations). *Nyt tidsskrift Matematik*, 20:33–39, 1909.

[FAK+12] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out

workloads on modern hardware. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, volume 47, pages 37–48. ACM, 2012.

[FJV+12] Benjamin Farley, Ari Juels, Venkatanathan Varadarajan, Thomas Ristenpart, Kevin D Bowers, and Michael M Swift. More for your money: exploiting performance heterogeneity in public clouds. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC)*, pages 20–33. ACM, 2012.

[FSYM13] Seyed Kaveh Fayazbakhsh, Vyas Sekar, Minlan Yu, and Jeffrey C Mogul. Flowtags: Enforcing network-wide policies in the presence of dynamic middlebox actions. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, pages 19–24. ACM, 2013.

[GBK14] Fabian Gorsler, Fabian Brosig, and Samuel Kounev. Performance queries for architecture-level performance models. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 99–110. ACM, 2014.

[GCGV06] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing performance isolation across virtual machines in Xen. In *Proceedings of the 7th ACM/IFIP/USENIX International Middleware Conference (Middleware)*, pages 342–362, 2006.

[GDM+13] HaiBing Guan, YaoZu Dong, RuHui Ma, Dongxiao Xu, Yang Zhang, and Jian Li. Performance enhancement for network I/O virtualization with efficient interrupt coalescing and virtual receive-side scaling. *Parallel and Distributed Systems, IEEE Transactions on*, 24(6):1118–1128, 2013.

[GK06] Pawel Gepner and Michal F Kowalik. Multi-core processors: New way to achieve high system performance. In *Proceedings of the International Symposium on Parallel Computing in Electrical Engineering (PARELEC)*, pages 9–13. IEEE, 2006.

[GLKS11] Sriram Govindan, Jie Liu, Aman Kansal, and Anand Sivasubramaniam. Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC)*, page 22. ACM, 2011.

[GNS11a] Vishal Gupta, Ripal Nathuji, and Karsten Schwan. An analysis of power reduction in datacenters using heterogeneous chip multiprocessors. *ACM SIGMETRICS Performance Evaluation Review*, 39(3):87–91, 2011.

[GNS11b] Vishal Gupta, Ripal Nathuji, and Karsten Schwan. An analysis of power reduction in datacenters using heterogeneous chip multiprocessors. *ACM SIGMETRICS Performance Evaluation Review*, 39(3):87–91, 2011.

[GSA+11] Ajay Gulati, Ganesha Shanmuganathan, Irfan Ahmad, Carl Waldspurger, and Mustafa Uysal. Pesto: online storage performance management in virtualized datacenters. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC)*, pages 19–33. ACM, 2011.

[Har03] Peter G. Harrison. Turning back time in Markovian process algebra. *Journal of Theoretical Computer Science*, 290:1947–1986, Jan. 2003.

[HBB12] Ashif S Harji, Peter A Buhr, and Tim Brecht. Comparing high-performance multi-core web-server architectures. In *Proceedings of the 5th Annual International Systems and Storage Conference (SYSTOR)*, pages 1–12. ACM, 2012.

[HBvR+13] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C Li. An analysis of Facebook photo caching. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 167–181. ACM, 2013.

[HKAC13] Raoufehsadat Hashemian, Diwakar Krishnamurthy, Martin Arlitt, and Niklas Carlsson. Improving the scalability of a multi-core web server. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 161–172. ACM, 2013.

[HKZ+11] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, volume 11, pages 22–22, 2011.

[HLP09] Peter G Harrison, Catalina M Lladó, and Ramon Puigjaner. A unified approach to modelling the performance of concurrent systems. *Simulation Modelling Practice and Theory*, 17(9):1445–1456, 2009.

[HP92] Peter G Harrison and Naresh M Patel. *Performance Modelling of Communication Networks and Computer Architectures*. Addison-Wesley Longman Publishing Co., Inc., 1992.

[HvQHK11] Nikolaus Huber, Marcel von Quast, Michael Hauck, and Samuel Kounev. Evaluating and Modeling Virtualization Performance Overhead for Cloud Environments. In *Proceedings of the International Conference on Cloud Computing and Services Science (CLOSER)*, pages 563–573, 2011.

[HW90] J Michael Harrison and Ruth J Williams. On the quasireversibility of a multiclass Brownian service station. *The Annals of Probability*, pages 1249–1268, 1990.

[IDC09] Waheed Iqbal, Matthew Dailey, and David Carrera. SLA-driven adaptive resource management for web applications on a heterogeneous compute cloud. *Cloud Computing*, pages 243–253, 2009.

[JJ09] Hye-Churn Jang and Hyun-Wook Jin. MiAMI: Multi-core aware processor affinity for TCP/IP over multiple network interfaces. In *Proceedings of the 17th IEEE Annual Symposium on High-Performance Interconnects (HOTI)*, pages 73–82. IEEE, 2009.

[KCV11] George Kousiouris, Tommaso Cucinotta, and Theodora Varvarigou. The effects of scheduling, workload type and consolidation scenarios on virtual machine perfor-

mance and their prediction through optimized artificial neural networks. *Journal of Systems and Software*, 84(8):1270–1291, 2011.

[KDS09] W J Knottenbelt, Nicholas J Dingle, and Tamas Suto. Chapter 9 Performance Trees : A Query Specification Formalism for Quantitative Performance Analysis. *Parallel, Distributed and Grid Computing for Engineering*, 2009.

[Ken53] David G Kendall. Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded Markov chain. *The Annals of Mathematical Statistics*, pages 338–354, 1953.

[KEY13] Shin-gyu Kim, Hyeonsang Eom, and Heon Y Yeom. Virtual machine consolidation based on interference modeling. *The Journal of Supercomputing*, 66(3):1489–1506, 2013.

[KHK+14] Jóakim V Kistowski, Nikolas Herbst, Samuel Kounev, et al. Limbo: a tool for modeling variable load intensities. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 225–226. ACM, 2014.

[KKEY12] Seungmin Kang, Shin-gyu Kim, Hyeonsang Eom, and Heon Y Yeom. Towards workload-aware virtual machine consolidation on cloud platforms. In *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication (IMCOM)*, page 45. ACM, 2012.

[Kle64] Leonard Kleinrock. Analysis of a time-shared processor. *Naval research logistics quarterly*, 11(1):59–73, 1964.

[Kle67] Leonard Kleinrock. Time-shared systems: A theoretical treatment. *Journal of the ACM (JACM)*, 14(2):242–261, 1967.

[Kle16] Martin Kleppmann. *Making sense of stream processing.* O'Reily Media, Inc., 2016.

[KMHK12] Melanie Kambadur, Tipp Moseley, Rick Hank, and Martha A Kim. Measuring interference between live datacenter applications. In *Proceedings of the International*

Conference on High Performance Computing, Networking, Storage and Analysis (SC), pages 51–62. IEEE, 2012.

[KTD12] Nontokozo P Khanyile, Jules-Raymond Tapamo, and Erick Dube. An analytic model for predicting the performance of distributed applications on multicore clusters. *IAENG International Journal of Computer Science*, 39:312–320, 2012.

[Lam77] Simon S. Lam. Queuing networks with population size constraints. *IBM Journal of Research and Development*, 21(4):370–378, 1977.

[LDDT12] Hui Lv, Yaozu Dong, Jiangang Duan, and Kevin Tian. Virtualization challenges: a view from server consolidation perspective. In *ACM SIGPLAN Notices*, volume 47, pages 15–26. ACM, 2012.

[LDH+09] Abigail S Lebrecht, Nicholas J Dingle, Peter G Harrison, William J Knottenbelt, and Soraya Zertal. Using bulk arrivals to model I/O request response time distributions in zoned disks and RAID systems. In *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS)*, pages 23–32. ACM/EAI conference series, 2009.

[LHK+12] Seung-Hwan Lim, Jae-Seok Huh, Youngjae Kim, Galen M Shipman, and Chita R Das. D-factor: a quantitative model of application slow-down in multi-resource shared systems. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, volume 40, pages 271–282. ACM, 2012.

[LWJ+13] Jie Li, Qingyang Wang, Danushka Jayasinghe, Junhee Park, Tao Zhu, and Calton Pu. Performance overhead among three hypervisors: An experimental study using hadoop benchmarks. In *Proceedings of the IEEE International Congress on Big Data (BigData Congress)*, pages 9–16. IEEE, 2013.

[LZK+11] Ang Li, Xuanran Zong, Srikanth Kandula, Xiaowei Yang, and Ming Zhang. CloudProphet: towards application performance prediction in cloud. In *Proceedings of the ACM SIGCOMM Conference*, volume 41, pages 426–427. ACM, 2011.

[MLPS10] Yiduo Mei, Ling Liu, Xing Pu, and Sankaran Sivathanu. Performance measurements and analysis of network i/o applications in virtualized cloud. In *Proceedings of the 3rd International Conference on Cloud Computing (CLOUD)*, pages 59–66. IEEE, 2010.

[MST+05] Aravind Menon, Jose Renato Santos, Yoshio Turner, G John Janakiraman, and Willy Zwaenepoel. Diagnosing performance overheads in the Xen virtual machine environment. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE)*, pages 13–23. ACM, 2005.

[MYM+11] Jayaram Mudigonda, Praveen Yalagandula, Jeff Mogul, Bryan Stiekes, and Yanick Pouffary. NetLord: a scalable multi-tenant network architecture for virtualized datacenters. In *Proceedings of the ACM SIGCOMM Conference*, number 4, pages 62–73. ACM, 2011.

[NBKR13] Qais Noorshams, Dominik Bruhn, Samuel Kounev, and Ralf Reussner. Predictive performance modeling of virtualized storage systems using optimized statistical regression techniques. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 283–294. ACM, 2013.

[NKG10] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys)*, pages 237–250. ACM, 2010.

[NLY+11] Jun Nakajima, Qian Lin, Sheng Yang, Min Zhu, Shang Gao, Mingyuan Xia, Peijie Yu, Yaozu Dong, Zhengwei Qi, Kai Chen, et al. Optimizing virtual machines using hybrid virtualization. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 573–578. ACM, 2011.

[NSG+13] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proceedings of the USENIX International Conference on Automated Computing (ICAC)*, 2013.

[OWZS13] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 69–84. ACM, 2013.

[PBYC13] Achille Peternier, Walter Binder, Akira Yokokawa, and Lydia Chen. Parallelism profiling and wall-time prediction for multi-threaded applications. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 211–216. ACM, 2013.

[PJD04] Ravi Prasad, Manish Jain, and Constantinos Dovrolis. Effects of interrupt coalescence on network measurements. *Passive and Active Network Measurement*, pages 247–256, 2004.

[PLH+15] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: a framework for NFV applications. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, pages 121–136. ACM, 2015.

[PLM+13] Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun Koh, Calton Pu, and Yuanda Cao. Who is your neighbor: Net I/O performance interference in virtualized clouds. *IEEE Transactions on Services Computing*, 6(3):314–329, 2013.

[PN16] Hossein Pishro-Nik. Introduction to probability, statistics and random processes. `https://www.probabilitycourse.com/chapter11/11_2_1_introduction.php`, Accessed 27 May 2016.

[RBG12] Anshul Rai, Ranjita Bhagwan, and Saikat Guha. Generalized resource allocation for the cloud. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC)*, pages 15–28. ACM, 2012.

[RKG+13] Alan Roytman, Aman Kansal, Sriram Govindan, Jie Liu, and Suman Nath. PAC-Man: Performance Aware Virtual Machine Consolidation. In *Proceedings of the*

4th ACM/SPEC International Conference on Performance Engineering (ICPE), pages 83–94. ACM, 2013.

[RNMV14] Navaneeth Rameshan, Leandro Navarro, Enric Monte, and Vladimir Vlassov. Stay-away, protecting sensitive applications from performance interference. In *Proceedings of the 15th International Middleware Conference (Middleware)*, pages 301–312. ACM, 2014.

[Roh15] Matthias Rohr. *Workload-sensitive Timing Behavior Analysis for Fault Localization in Software Systems*. BoD–Books on Demand, 2015.

[RTG+12] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC)*, pages 56–69. ACM, 2012.

[SBK06] Tamas Suto, Jeremy T Bradley, and William J Knottenbelt. Performance trees: A new approach to quantitative performance specification. In *Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 303–313. IEEE, 2006.

[SIB+14] Sahil Suneja, Canturk Isci, Vasanth Bala, Eyal De Lara, and Todd Mummert. Non-intrusive, Out-of-band and Out-of-the-box Systems Monitoring in the Cloud. In *Proceedings of International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, volume 42, pages 249–261. ACM, 2014.

[SSGW11] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. CloudScale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC)*, pages 5–18. ACM, 2011.

[SSM+11] Akbar Sharifi, Shekhar Srikantaiah, Asit K Mishra, Mahmut Kandemir, and Chita R Das. METE: meeting end-to-end QoS in multicores through system-wide resource management. In *Proceedings of the ACM SIGMETRICS International*

Conference on Measurement and Modeling of Computer Systems, pages 13–24. ACM, 2011.

[SST12] Upendra Sharma, Prashant Shenoy, and Donald F Towsley. Provisioning multi-tier cloud applications using statistical bounds on sojourn time. In *Proceedings of the 9th International Conference on Autonomic Computing (ICAC)*, pages 43–52. ACM, 2012.

[SWWL14] Ryan Shea, Feng Wang, Haiyang Wang, and Jiangchuan Liu. A deep investigation into network performance in virtual machine based cloud environments. In *Proceedings of IEEE Conference on Computer Communications (INFOCOM)*, pages 1285–1293. IEEE, 2014.

[TB14] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. Prentice Hall Press, 2014.

[TBO+13] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. IOFlow: a software-defined storage architecture. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 182–196. ACM, 2013.

[TCGK12] Alexey Tumanov, James Cipar, Gregory R Ganger, and Michael A Kozuch. alsched: Algebraic scheduling of mixed workloads in heterogeneous clouds. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC)*, page 25. ACM, 2012.

[TGS14] Priyanka Tembey, Ada Gavrilovska, and Karsten Schwan. Merlin: Application-and platform-aware resource allocation in consolidated server systems. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pages 1–14. ACM, 2014.

[TIIN10] Omesh Tickoo, Ravi Iyer, Ramesh Illikkal, and Don Newell. Modeling virtual machine performance: challenges and approaches. *ACM SIGMETRICS Performance Evaluation Review*, 37(3):55–60, 2010.

[TK14] Howard M Taylor and Samuel Karlin. *An introduction to stochastic modeling.* Academic press, 2014.

[TT13] Bogdan Marius Tudor and Yong Meng Teo. On understanding the energy consumption of arm-based multicore servers. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, number 1, pages 267–278. ACM, 2013.

[TZP+16] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*, pages 35–48. ACM, 2016.

[UWH+15] Rahul Urgaonkar, Shiqiang Wang, Ting He, Murtaza Zafer, Kevin Chan, and Kin K Leung. Dynamic Service Migration and Workload Scheduling in Micro-Clouds. In *Proceedings of the 33rd International Symposium on Computer Performance, Modeling, Measurements and Evaluation (IFIP WG 7.3 Performance)*, 2015.

[Var16] Various. AWS IP Address Ranges. `http://docs.aws.amazon.com/general/latest/gr/aws-ip-ranges.html`. Accessed 28 May 2016.

[VF07] Bryan Veal and Annie Foong. Performance scalability of a multi-core web server. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS)*, pages 57–66. ACM, 2007.

[Vir16] Jorma Virtamo. PS queue. `http://www.netlab.tkk.fi/opetus/s383141/kalvot/E_psjono.pdf`. Accessed 26 May 2016.

[VN10] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC)*, pages 193–204. ACM, 2010.

[WCB07] Wenji Wu, Matt Crawford, and Mark Bowden. The performance analysis of Linux networking–packet receiving. *Computer Communications*, 30(5):1044–1057, 2007.

[WGB+10] David Wentzlaff, Charles Gruenwald III, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal. An operating system for multicore and clouds: mechanisms and implementation. In *Proceedings of the 1st ACM Symposium on Cloud computing (SoCC)*, pages 3–14. ACM, 2010.

[WID+14] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas JA Harvey, and Andrew Warfield. Characterizing storage workloads with counter stacks. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 335–349, 2014.

[WSL12] Fetahi Wuhib, Rolf Stadler, and Hans Lindgren. Dynamic resource allocation with management objectives - Implementation for an OpenStack cloud. In *Proceedings of the 8th Internationl Conference on Network and Service Management (CNSM)*, pages 309–315. IEEE, 2012.

[WZY+13] Weina Wang, Kai Zhu, Lei Ying, Jian Tan, and Li Zhang. A throughput optimal algorithm for map task scheduling in mapreduce with data locality. *ACM SIGMETRICS Performance Evaluation Review*, 40(4):33–42, 2013.

[XBNJ13] Yunjing Xu, Michael Bailey, Brian Noble, and Farnam Jahanian. Small is better: Avoiding latency traps in virtualized data centers. In *Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC)*, pages 7–20. ACM, 2013.

[YHJ+10] Kejiang Ye, Dawei Huang, Xiaohong Jiang, Huajun Chen, and Shuang Wu. Virtual machine based energy-efficient data center architecture for cloud computing: a performance perspective. In *Proceedings of the IEEE/ACM International Conference on Green Computing and Communications (GreenCom)*, pages 171–178. IEEE, 2010.

[ZCM11] Eyal Zohar, Israel Cidon, and Osnat Ossi Mokryn. The power of prediction: Cloud bandwidth and cost reduction. In *Proceedings of the ACM SIGCOMM Conference*, number 4, pages 86–97. ACM, 2011.

[ZCW+13] Qing Zheng, Haopeng Chen, Yaguang Wang, Jian Zhang, and Jiangang Duan. COSBench: cloud object storage benchmark. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 199–210. ACM, 2013.

[ZT12] Qian Zhu and Teresa Tung. A performance interference model for managing consolidated workloads in QoS-aware clouds. In *Proceedings of the 5th International Conference on Cloud Computing (CLOUD)*, pages 170–179. IEEE, 2012.

[ZTH+13] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. CPI 2: CPU performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*, pages 379–391. ACM, 2013.