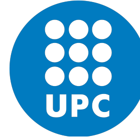


Co-funded by the
Erasmus+ Programme
of the European Union

On the Role of Performance Interference in Consolidated Environments

NAVANEETH RAMESHAN

School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden 2016
and
Facultat d'Informática de Barcelona
Universitat Politécnica de Catalunya
Computer Networks and Distributed Systems Group (CNDS)
Departament d'Arquitectura de Computadors (DAC)
Barcelona, Spain 2016

KTH School of Information and
Communication Technology
SE-164 40 Kista
SWEDEN

TRITA-ICT 2016:28
ISBN 978-91-7729-113-8

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av doktorsexamen i Informations- och kommunikationsteknik måndagen den 24 oktober 2016 klockan 12:00 i Sal C, Electrum, Kungl Tekniska högskolan, Kistagången 16, Kista.

© Navaneeth Rameshan, October 2016

Tryck: Universitetsservice US AB

Abstract

With the advent of resource shared environments such as the Cloud, virtualization has become the de facto standard for server consolidation. While consolidation improves utilization, it causes performance-interference between Virtual Machines (VMs) from contention in shared resources such as CPU, Last Level Cache (LLC) and memory bandwidth. Over-provisioning resources for performance sensitive applications can guarantee Quality of Service (QoS), however, it results in low machine utilization. Thus, assuring QoS for performance sensitive applications while allowing co-location has been a challenging problem. In this thesis, we identify ways to mitigate performance interference without undue over-provisioning and also point out the need to model and account for performance interference to improve the reliability and accuracy of elastic scaling. The end goal of this research is to leverage on the observations to provide efficient resource management that is both performance and cost aware. Our main contributions are threefold; first, we improve the overall machine utilization by executing best-effort applications along side latency critical applications without violating its performance requirements. Our solution is able to dynamically adapt and leverage on the changing workload/phase behaviour to execute best-effort applications without causing excessive interference on performance; second, we identify that certain performance metrics used for elastic scaling decisions may become unreliable if performance interference is unaccounted. By modelling performance interference, we show that these performance metrics become reliable in a multi-tenant environment; and third, we identify and demonstrate the impact of interference on the accuracy of elastic scaling and propose a solution to significantly minimise performance violations at a reduced cost.

Resumen

Con la aparición de entornos con recurso compartidos tales como la nube, la virtualización se ha convertido en el estándar de facto para la consolidación de servidores. Mientras que la consolidación mejora la utilización, también causa interferencia en el rendimiento de las máquinas virtuales (VM) debido a la contención en recursos compartidos, tales como CPU, el último nivel de caché (LLC) y el ancho de banda de memoria. El exceso de aprovisionamiento de recursos para aplicaciones sensibles al rendimiento puede garantizar la calidad de servicio (QoS), sin embargo, resulta en una baja utilización de la máquina. Por lo tanto, asegurar QoS en aplicaciones sensibles al rendimiento, al tiempo que permitir la co-localización ha sido un problema difícil. En esta tesis, se identifican las formas de mitigar la interferencia sin necesidad de sobre-aprovisionamiento y también se señala la necesidad de modelar y contabilizar la interferencia en el desempeño para mejorar la fiabilidad y la precisión del escalado elástico. El objetivo final de esta investigación consiste en aprovechar las observaciones para proporcionar una gestión eficiente de los recursos considerando tanto el rendimiento como el coste. Nuestras contribuciones principales son tres; primero, mejoramos la utilización total de la máquina mediante la ejecución de aplicaciones best-effort junto con aplicaciones críticas en latencia sin vulnerar sus requisitos de rendimiento. Nuestra solución es capaz de adaptarse de forma dinámica y sacar provecho del comportamiento cambiante de la carga de trabajo y sus cambios de fase para ejecutar aplicaciones best-effort, sin causar interferencia excesiva en el rendimiento; segundo, identificamos que ciertos parámetros de rendimiento utilizados para las decisiones de escalado elástico pueden no ser fiables si no se tiene en cuenta la interferencia en el rendimiento. Al modelar la interferencia en el rendimiento, se muestra que estas métricas de rendimiento resultan fiables en un entorno multi-proveedor; y tercero, se identifica y muestra el impacto de la interferencia en la precisión del escalado elástico y se propone una solución para minimizar significativamente vuneraciones de rendimiento con un coste reducido.

Sammanfattning

I och med införandet av resursdelande miljöer som t.ex. med Cloud-tjänster, har virtualisering blivit de facto standard för konsolidering av servrar. Medan konsolidering förbättrar utnyttjandet , orsakar det prestanda-interferens mellan virtuella maskiner (VM) genom konflikter om delade resurser som CPU-er, sista nivåns cache (”Last-level-cache”, LLC) och minnesbandbredd. Att erbjuda överkapacitet av resurser för prestandakänsliga tillämpningar kan garantera servicekvaliteten (“Quality of Service”, QoS), men det resulterar i lågt utnyttjande av hårdvaran. Så, att garantera QoS för prestandakänsliga tillämpningar medan man samtidigt tillåter “co-location” har varit ett utmanande problem. I denna avhandling identifierar vi olika sätt att hantera prestandainterferens utan att erbjuda onödig överkapacitet, och vi pekar också ut behovet att modellera och ta i beaktande prestandainterferens för att förbättra pålitligheten och precisionen i elastisk skalning. Slutmålet för denna forskning är att dra nytta av observationerna för att erbjuda effektiv resursmanagement som tar hänsyn till både prestanda och kostnad. Våra viktigaste bidrag är tre; först förbättrar vi utnyttjandegraden för hårdvaran genom att exekvera “best-effort”-tillämpningar sida vid sida med latenskritiska tillämpningar utan att påverka prestandakraven. Vår lösning kan dynamiskt anpassa sig till och dra nytta av den föränderliga arbetslasten/fasbeteendet för att exekvera “best-effort”-tillämpningar utan att orsaka överdrivna interferenser i prestanda; för det andra så identifierar vi att vissa prestandametriker som används för elastiska skalningsbeslut kan komma att bli opålitliga om prestanda interfererar okontrollerat. Genom att modellera prestandainterferenser, så kan vi visa att dessa prestandametriker är pålitliga i en fleranvändarmiljö (“multi-tenant environment”); och för det tredje så identifierar vi och demonstrerar effekten av interferens på exaktheten i den elastiska skalningen och föreslår en lösning för att signifikant minimera prestandastörningarna med reducerad kostnad.

Acknowledgements

This thesis is made under joint supervision and published in two languages and the work was supported in part by the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) funded by the Education, Audiovisual and Culture Executive Agency (EACEA) of the European Commission under the FPA 2012-0030.

I'm greatly indebted to Leandro Navarro and owe a lot to him for making this PhD journey significantly easier and for all the advice and assistance he has offered, even if it meant going out of his way. His selflessness and humility is beyond what I have seen in anyone else. I consider myself fortunate to have had the opportunity to know him and will strive to inculcate some of his values in my everyday life.

I'd also like to sincerely thank Vladimir Vlassov for making my stay at KTH pleasurable and for all the discussions and advice he has given throughout the course of this PhD. His actions have always been in the best of interest for me and I couldn't have cruised through this journey without his help.

I also extend my sincere thanks to Manish Parashar, Omer Rana, Rosa Badia and Ramin Yahyapour for their valuable feedback and for being a part of the committee. It was truly an honour to have had you on my committee.

I can't thank enough Ying, Leila, Enric Monte, Lydia, Robert Birke and Jordi Guitart for all the technical discussions and brain storming sessions that have been instrumental in shaping this thesis. To my friends Kiarash, Lalith, Mario, Nick, Ying, Leila, Vasia, Davis, Maria, Gurjinder, Umar, Manos, João, Vamis, Leonardo, Amin, Ester, Davide, Veena, Saranya, and Menan whose presence helped me stay motivated throughout the course of this PhD and were constantly uplifting. To UPC colleagues for those much deserved ping-pong breaks in the lab.

To my parents and my sister for their unconditional love, unfettered support, understanding and sacrifice. I'm forever indebted to you.

To Veena, for her love and consistent moral support. Her frequent phone conversations, without which, this thesis in its current form would have been completed at least 6 months back!

Contents

List of Figures **xiii**

List of Publications **xix**

1 Introduction **1**

1.1 Performance vs. Utilization 2

1.2 Thesis Overview 3

2 Background and Related Work **5**

2.1 Background 5

2.2 Related Work 7

3 Contributions **15**

3.1 Performance Interference from a single host perspective 15

3.2 Performance Interference from a distributed system perspective 17

4 Dynamic Reconfiguration to minimise Interference **21**

4.1 Introduction 22

4.2 Background 23

4.3 Stay-Away Mechanism 26
4.4 Optimisations and Overhead 35
4.5 Scalability 36
4.6 Template Properties 37
4.7 Evaluation 38
4.8 Summary 46

5 Reliable Elastic Scaling Decisions 49

5.1 Introduction 50
5.2 Elastic Scaling 51
5.3 Motivation 54
5.4 Experimental Analysis 55
5.5 System Overview 62
5.6 Characterising Contention 65
5.7 Elasticity Controller 71
5.8 Experimental Evaluation 75
5.9 Related Work 81
5.10 Summary 83

6 Augment Elastic Scaling 85

6.1 Introduction 86
6.2 Problem Definition 88
6.3 Experimental Analysis 90
6.4 Solution Overview 92

6.5 Middleware Interface to Quantify Capacity (MI) 94
6.6 Augmentation Engine 99
6.7 Experimental Evaluation 102
6.8 Summary 108

7 Conclusions 109

Bibliography 113

List of Figures

1.1 Trade-off between performance and utilization 2

4.1 Total Workload variation of Wikipedia during the period 1/1/2011 to 5/1/2011 22

4.2 Stay-Away mechanism. This figure illustrates the 3 steps: Mapping, Predicting Transition and Action. The darker circle on the top of the map represents a QoS violation. 26

4.3 State-space exploration 29

4.4 Variation of the radius of *violation-range* as distance between the *violation-state* and nearest *safe-state* varies 30

4.5 All 4 execution modes when VLC streaming is co-located with Soplex from SPEC CPU 2006 32

4.6 Snapshot of instantaneous transition of states when VLC transcoding is co-located with CPUBomb in the mapped space. Action status:False indicates that Stay-Away was not throttling the batch application during the snapshot 39

4.7 Snapshot of gradual transition of states when VLC streaming is co-located with Twitter-Analysis in the mapped space. Action status:True indicates that the batch application was being throttled during the snapshot 40

4.8 QoS violations of VLC streaming and utilization gained by co-locating VLC streaming alongside CPUBomb 41

4.9 QoS violations of VLC streaming and utilization gained by co-locating VLC streaming alongside Twitter-Analysis 42

4.10 Gained Utilization when Webservice is co-located with different Batch Applications 43

4.11 QoS of Webservice with a mix of CPU and Memory intensive workload when co-located with different Batch Applications . . 43

4.12 QoS of Webservice with CPU intensive workload when co-located with different Batch Applications 44

4.13 QoS of Webservice with Memory intensive workload when co-located with different Batch Applications 44

4.14 The colour gradient for Webservice is a measure of the stress on its performance. Darker colour indicates higher stress. Dark colour bands for Twitter-Analysis represents period of its execution and lighter colour bands represents the period it is throttled. 45

4.15 Template with CPUBomb 46

4.16 VLC with soplex 47

5.1 Architecture of a typical elastic scaling process 52

5.2 Variation of latency over time for a constant workload (RPS). Until 10 mins, the application runs in isolation. After 10 mins, other applications are executed on the physical host, generating contention at the shared system resources. 54

5.3 CPU utilization (%) vs. Workload intensity (RPS) for Memcached with and without co-location for different configurations. Config 3 brings down the unpredictability solely to the intensity in contention and the number of co-located VMs. 56

5.4 Variation of Latency (ms) vs. CPU utilization (%) in Memcached with and without co-location for different configurations. Config 3 brings down the unpredictability solely to the intensity in contention and the number of co-located VMs. 58

5.5 Variation of Latency (ms) versus Workload (RPS) in Memcached with and without co-location for different configurations. Config 3 brings down the unpredictability solely to the intensity in contention and the number of co-located VMs. 59

5.6 Frequency-scaling affects performance predictability. Once frequency-scaling is controlled, performance variation from interference becomes predictable as shown by blue line. 61

5.7 Architecture of the Middleware 63

5.8 Configurations for generating contention at different resources. S denotes the storage system and M(S) denotes the memory allocation of S. C denotes the co-runners and M(C) denotes the memory allocation of C. I denotes the core serving interrupts. . 65

5.9 The drop in performance of Memcached for different throughputs. Memcached is run alongside 6 instances of different co-runners. 66

5.10 The drop in performance of Redis for different throughputs. Redis is run alongside 6 instances of different co-runners. 67

5.11 Interference-index quantifies the performance drop suffered by Memcached based on the behaviour of the co-runner. 71

5.12 Throughput Performance Model for different levels of Interference. Red and green points mark the detailed profiling region of SLO violation and safe operation respectively in the case of no interference. 74

5.13 Experimental setup. The workload and interference are divided into 4 phases of different combinations demarcated by vertical lines. (b) is the interference index generated when running Memcached and (c) is the interference index generated when running Redis. 77

5.14 Results of running Memcached across the different phases. (a) and (b) shows the number of VMs and latency of Memcached for a workload based model. (c) and (d) hows the number of VMs and latency of Memcached for a CPU based model. 78

5.15 Results of running Redis across the different phases. (a) and (b) shows the number of VMs and latency of Redis for a workload based model. (c) and (d) hows the number of VMs and latency of Redis for a CPU based model. 79

5.16 Utility measure for different Scaling approaches. VMs_ideal represents the theoretically best scaling possible without any over-provisioning or SLO violations. VMs_c_hubbub and VMs_c_standard represents the utility measure of CPU based scaling using Hubbub and standard modelling respectively. VMs_hubbub and VMs_standard represents the utility measure of workload based scaling using Hubbub and standard modelling respectively. . . 81

6.1 Memcached service experiencing a long period of SLO violation during periods or scaling out and scaling down because of unmet capacity from performance interference. 89

6.2 VM capacity reduces with increasing performance interference. In this example the SLO is set to 1.8 ms. 91

6.3 (a) Impact of interference for different load on Memcached. Interference impacts performance only at higher loads. (b) Time taken to load Memcached data on a AWS large instance for different data sizes. 92

6.4 Architecture of the system components 93

6.5 Unmet capacity from figure 6.1 pruned by augmenting the elasticity controller to detect and quantify interference. 95

6.6 The drop in performance of Memcached for contention at different levels of memory subsystem. Memcached is run alongside multiple instances of different co-runners. 96

6.7 Interference-index quantifies the performance drop suffered by Memcached based on the behaviour of the co-runner. 98

6.8 Results demonstrating load balancer reconfiguration by AE. Server timeline shows the status of the server over time. White indicates that there is no instance on the server. Blue indicates that the instance on the server is preparing the data. Gradient of red indicates the amount of interference on the server. Dark red indicates high interference 104

6.9 Results demonstrating convergence time with AE and without AE. Server timeline shows the status of the server over time. White indicates that there is no instance on the server. Blue indicates that the instance on the server is preparing the data. Gradient of red indicates the amount of interference on the server. Dark red indicates high interference 106

6.10 Results demonstrating informed scaling down with AE. Server timeline shows the status of the server over time. White indicates that there is no instance on the server. Blue indicates that the instance on the server is preparing the data. Gradient of red indicates the amount of interference on the server. Dark red indicates high interference 107

6.11 108

List of Publications

[P1] Navaneeth Rameshan, Ying Liu, Leandro Navarro and Vladimir Vlassov. **Augmenting Elasticity Controllers for Improved Accuracy** *13th IEEE/USENIX International Conference on Autonomic Computing (ICAC) (Best Paper Candidate)*, July 2016.

[P2] Navaneeth Rameshan, R. Birke, V. Vlassov, L. Navarro, B. Urgaonkar, G. Kesidis, M. Schmatz and L. Y. Chen. **Profiling Memory Vulnerability of Big-Data Applications** *46th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Industrial track*, June 2016.

[P3] Navaneeth Rameshan, Ying Liu, Vladimir Vlassov and Leandro Navarro. **Elastic Scaling: A Multi-Tenant Perspective** *6th international workshop on Big Data and Cloud Performance (DCPerf), Co-located with ICDCS (Invited Paper)*, June 2016.

[P4] Navaneeth Rameshan, Ying Liu, Leandro Navarro and Vladimir Vlassov. **Hubbub-Scale: Towards Reliable Scaling under Multi-Tenancy** *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2016.

[P5] Navaneeth Rameshan Leandro Navarro, Enric Monte and Vladimir Vlassov. **Stay-Away, Protecting Sensitive Applications from Performance-Interference** *15th ACM/IFIP/USENIX International Middleware Conference (Middleware)*, December 2014.

[P6] Navaneeth Rameshan, Ioanna Tsalochidou and Leandro Navarro. **A Monitoring System for Community-lab** *The 16th ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM) (Best Demo Award)*, November 2013.

Other Publications

[P7] Ying Liu, Navaneeth Rameshan, Enric Monte, Vladimir Vlassov and Leandro Navarro. **ProRenaTa: Proactive and Reactive Tuning to Scale a Distributed Storage System** *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2015.

[P8] L. Sherifi, Navaneeth Rameshan , Freitag, F., and Veiga, L. **Energy efficiency dilemma: p2p-cloud vs. mega-datacenter** *IEEE 6th International Conference on Cloud Computing Technology and Science (CloudCom) (Best Paper Nomination)*, December 2014.

[P9] Simao Jose, Navaneeth Rameshan, and Luis Veiga. **Resource-Aware Scaling of Multi-threaded Java Applications in Multi-tenancy Scenarios** *IEEE 5th International Conference on Cloud Computing Technology and Science (CloudCom)*, December 2013.

Chapter 1

Introduction

In recent years, there has been an increasing amount of growth in large scale distributed computing infrastructures aimed at satisfying diverse goals. The increasing shift toward server side computing has resulted in a class of computing systems called the warehouse-scale computers (WSCs). They typically comprise of a massive scale of software infrastructure, data repositories, and hardware platform and consists of multiple programs that interact with each other to provide a complex service. One such infrastructure is the cloud and marked the shift toward server-side computing. Cloud computing is widely used for offloading computing, primarily due to the cost benefits for both the end-users and the operators and other myriad of advantages it offers. Cloud providers such as Amazon EC2, Microsoft Azure and Google compute engine host tens of thousands of applications on a daily basis. Another class of such distributed infrastructure are large scale testbeds such as Planet-lab [1] and Community-lab [2], that aim to provide a platform for researchers to experiment with planetary-scale network services.

Although both these infrastructures differ in their purposes, they must satisfy a common goal of application isolation and resource efficiency. To achieve this, the infrastructure economics must allow servers to be shared among multiple users and at the same time guarantee operational isolation of applications. To this end, virtualization is the most widely adopted solution to guarantee these goals. It allows for the hardware infrastructure to be shared by various users, without giving direct access to the underlying hardware. Virtualization provides isolation and an illusion of dedicated hardware access to the users, while in reality the provider retains complete control of the underlying hardware

infrastructure. While Virtualization guarantees application isolation, better resource utilization and lower operational costs, it comes at the price of application slow down and performance degradation in ways that cannot be modelled or seen easily. In the next section, we explain this trade off.

1.1 Performance vs. Utilization

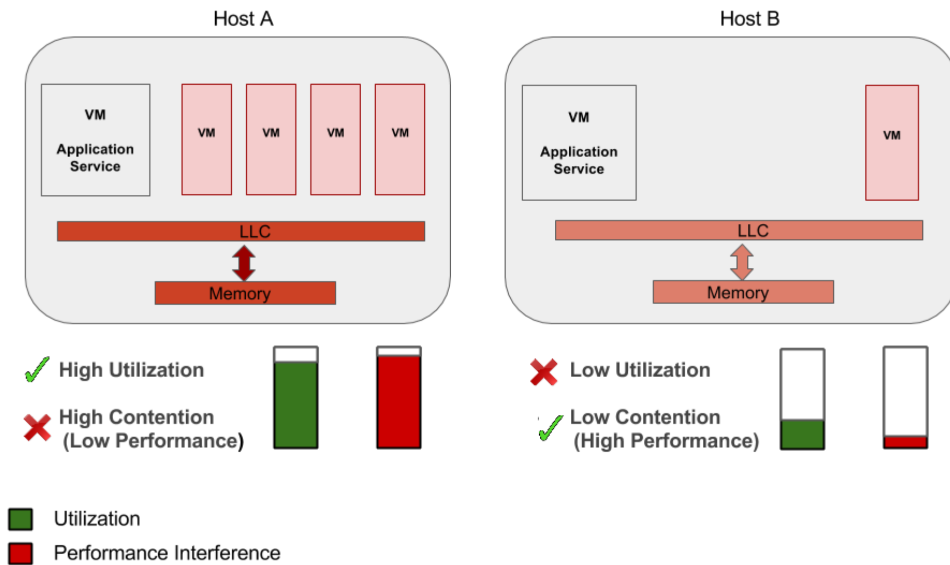


Figure 1.1: Trade-off between performance and utilization

In order for an infrastructure to be efficient and cost effective, it is important to keep the resources highly utilised. Virtualization has become the de facto standard to consolidate and improve resource utilization, as it allows for multiple users to share the available resources. Increased resource utilization can in turn reduce costs, server purchases and also minimise power consumption. However, increasing utilization alone is not a panacea as it involves a trade-off with performance. As more and more virtual machines (VM) are consolidated to increase utilization, they compete for shared resources and consequently degrade each others performance, commonly known as performance interference. Inter-VM performance interference happens when behavior of one VM adversely affects the performance of another due to contention in the shared resources in the system. For example, multiple VMs when consolidated on a physical host, will share the last level cache. If any of the VMs aggressively use the last level cache, it evicts the cached data of other VMs thereby degrading

the performance. Interference can happen at any level: CPU, memory, I/O buffer, processor cache etc. On the other hand, difficulties to model and predict performance interference has resulted in the heavy handed approach of disallowing any co-locations with performance sensitive applications, a major contributor to low machine utilization in data centers [3]. Figure 1.1 shows such a tradeoff. Host A is highly utilised but the contention from co-located VMs deteriorate the performance of the application service. Host B shows the other end of the spectrum where there is low contention on the shared resources but the system remains under-utilized. Therefore, to be efficient, it is important to strike a right balance between increasing utilization and guaranteeing performance.

1.2 Thesis Overview

This thesis, in whole, aims to improve the overall utilization of the resources while respecting the performance bounds of latency critical applications. It is divided into 2 parts, with the first part focusing on a single host and the second part focusing on distributed hosts. On a single host, this thesis explores dynamic reconfiguration to mitigate the impact of performance interference. Specifically, on a single host, we aim to improve utilization by co-locating latency critical application with best-effort batch applications. Performance requirements of the latency critical applications are then guaranteed at the expense of the batch applications by throttling them only when necessary.

In the second part, we investigate if there are consequences of performance interference that go beyond performance degradation. This includes service reliability and accuracy. With elastic scaling as a use case, we study how reliability and accuracy of scaling decisions are affected in the presence of performance interference. Driven by our observations, we finally demonstrate how both reliability and accuracy can be improved without undue over-provisioning, thereby guaranteeing performance.

Thesis organization

This thesis is organized as follows. In chapter 2 we build the necessary background for the reader to gauge the contribution, and present the state of the art on resource management techniques for dealing with interference.

Chapter 3 presents the main research questions addressed in this thesis and summarizes the main contributions to provide the reader a coherent view of the separate contributions detailed in chapters 4, 5 and 6. Finally, we present the conclusions and future work in chapter 7.

Chapter 2

Background and Related Work

In this chapter, we review background information and the related work with specific focus on the most recent work regarding resource management approaches for dealing with performance interference.

2.1 Background

Performance interference can happen at any level: CPU, memory, cache, I/O buffer etc. Resources themselves can be divided into partitionable and non-partitionable types. Partitionable resources are those that can be partitioned across multiple application for isolated access. CPU is one such partitionable resource. Although CPU can be virtualized and time shared between multiple applications, this can cause contention for CPU and degrade performance, depending on the scheduling algorithm for CPU sharing and the type of application. A simple way to mitigate such type of contention is to partition the available CPUs across different applications and provide isolated access to CPU. This is possible because CPU is a partitionable resource. There also exists utilities like nice [4] to manage application priorities to deal with CPU interference. On the other hand, there are resources such as the last level cache and memory controller that cannot be partitioned without special hardware support. Therefore, it is challenging to mitigate interference on such non-partitionable shared resources and requires careful consideration from the software side to minimise performance degradation.

Mitigating interference on non-partitionable shared resources can be approached from different perspectives and may involve different steps. For example, it can be seen as a VM placement problem. An immediate solution that seems feasible is to classify applications based on its major share of resource consumed and treat it as an optimal placement problem. For example, placing a memory intensive application with a CPU intensive application fares better than placing 2 memory intensive applications together. Although an application can have a dominant share of resource it consumes, it typically goes through multiple phases during the life cycle of its execution and can use multiple resources. Under such circumstances scheduling alone is insufficient to guarantee performance. Although scheduling can minimize resource contention, it does not entirely alleviate the problem. Another challenge with VM placement is the temporal overlap of the execution phases between different applications. Depending on how the temporal behaviours of applications overlap, performance can vary significantly.

An alternative approach to overcome the problems with scheduling is to combine dynamic re-configuration to further mitigate the limitations of changing phase behaviour and temporal overlap. Dynamic re-configuration involves techniques that adapt an application during run time. VM migration is one such technique. For example: when an application severely suffers from performance interference, it can be migrated to another host. There are other reconfiguration approaches possible such as resource scaling, dynamic frequency scaling etc. Cost of dynamic reconfiguration is also an important concern. If the benefits accrued by reconfiguration does not outweigh the cost involved, an adaptation can backfire.

If the goal of any of this approach is to maintain a desired level of performance it becomes imperative to detect the presence of performance interference and also quantify the same to know when/how much to reconfigure. Detecting performance interference can itself be challenging since an application's performance can drop either from contention, surge in workload or even from varying phase behaviour that may result in a resource bottleneck. On the other hand, measuring the intensity of interference is equally important to quantify the reconfiguration approach. For example, resource scaling can be used upon detecting interference but the exact amount of resource to be scaled depends on the level of interference. Aggressively scaling in the presence of light interference can starve the co-located application from making progress

and diffident scaling in the presence of high interference can be insufficient to guarantee performance. Thus, it is important to quantify interference so as to guarantee performance without being overly aggressive.

Another class of approach to mitigate interference involves resource partitioning at a fine grained level. On the memory subsystem, cache partitioning techniques are prevalent that aim to guarantee an isolated share of last level cache for different applications. There also exists hardware partitioning approaches that achieve this level of guarantees on the hardware. Intel recently released a set of Xeon machines with Cache Allocation Technology (CAT) [5] that guarantees cache space for any application at the hardware level. However, there have been no large scale deployments of such servers and we do not envision large scale hardware replacement in the near future.

In the next section, we highlight the most relevant related work in the context of scheduling, dynamic reconfiguration, interference detection and quantification. Most of the works overlap across multiple context and we present them in the category we deem most apt.

2.2 Related Work

2.2.1 Dynamic Reconfiguration

DejaVu [6] relies on a online-clustering algorithm to adapt to load variations by comparing the performance of a production VM and a replica of it that runs in a sand-box to detect interference. It mitigates interference by over-provisioning resources. Unfortunately, DejaVu has a high cost as it requires an additional sand-box for executing the replica. A similar system, DeepDive [7], first relies on a warning system running in the VMM to conduct early interference analysis. When the system suspects that one or more VMs are subjected to interference, it clones the VM on-demand and executes it in a sandboxed environment to detect interference by comparing the differences in relevant measured metrics. If interference exists, the most aggressive VM is migrated on to another physical machine. It incurs overhead in the form of cloning and migrating VMs. Migrating VMs is an expensive and time consuming operation.

Another approach to managing performance interference by measuring it in situ was presented by Nathuji et al. [8]. Their approach uses an online MIMO model to capture the performance interference effects in terms of resource allocations. The method then adjusts the processor allocation for each application based on the required performance level, in effect compensating for performance degradation by allocating additional resources. It achieves this by giving unallocated resources to an application to prevent falling below the QoS requirement. Q-Clouds improves performance as long as there is headroom available.

Lingjia et al. [9] present a compilation approach that identifies code regions that cause contention and transforms those regions to reduce their contentious nature. In essence, it recompiles the application to throttle down memory access rate to reduce interference on high priority applications.

Heracles [10] aims to leverage periods of low utilization when running latency-critical workloads to execute batch-applications. It uses a feedback-based controller that enables safe co-location of batch applications alongside latency-critical tasks. It dynamically manages multiple hardware and software isolation mechanisms to ensure performance guarantees for latency-critical applications. Specifically it relies on Cache Allocation Technology (CAT), scaling down the number of cores, DVFS and network bandwidth limiting to tune the performance of best-effort applications when they begin to degrade the performance of latency-critical applications. This is achieved through a hierarchical controller with sub-controllers for each resource (CPU, Memory and Network) that work in unison to decide which isolation mechanism to use.

iAware [11] focuses on how to make live migration of VMs interference aware. It empirically captures the essential relationship between performance interference and different metrics by executing benchmark workloads. It then jointly estimates and minimizes both migration and co-location interference by designing a simple multi-resource demand supply model.

Amiya et al. [12] guarantee performance of webservers through application reconfiguration techniques. They use cache-miss rate along with application reports on throughput and response time to detect interference. Upon detecting interference their solution tunes web server specific parameters such as MaxClients and KeepaliveTimeout to mitigate the detrimental effects of interference maintain the performance requirement.

2.2.2 Partitioning

Another class of work has also investigated providing QoS management for different applications on multicore by partitioning shared resources [13, 14, 15]. While demonstrating promising results, resource partitioning typically requires changes to the hardware design, which is not feasible for existing systems.

In Ubik [16] the authors show that guaranteeing average performance is not enough to maintain tail latency. They show that resources such as cache and cores exert inertia on the instantaneous workload performance, and these resources when shared degrades the tail latency significantly. While other works on cache partitioning can be adapted to provide capacity isolation and improve cache utilization, ignoring inertia sacrifices the performance of batch applications that are co-located with latency-critical applications. They propose a dynamic partitioning technique that predicts and exploits the transient behaviour of latency-critical workloads to dynamically manage cache allocations without degrading tail latencies.

2.2.3 Quantifying Degradation

Recent efforts [17, 18, 19] demonstrate that it is possible to accurately predict the degradation caused by interference by prior analysis of workload. However, in practice, applications are not available prior to their deployment and often run for a long time, so it may not feasible to perform this analysis for all kinds of application. Koh et al [20] propose a technique for predicting performance degradation of co-located applications based on their resource usage statistics. When a new application is hosted, its resource vector is compared with that of known applications and is mapped to the weighted average of one or more known applications whose resource vectors it closely resembles. Then the performance degradation of the new application is predicted based on already recorded performance degradation of the mapped/representative applications. It requires the need to have an already profiled application that closely resembles an incoming application and does not capture the temporal properties of the application.

In [21], the authors show that it is important to consider temporal properties (phases) when applications are scheduled. They show that applications can experience different amount of degradation depending on which phases overlap

between the scheduled applications. The authors propose a method for efficiently investigating the performance variability due to cache contention while taking into account phase behaviour. They capture input data from isolated execution of the application and is combined with a phase-aware performance model that quickly evaluates hundreds of overlappings. Their results show that taking into account the temporal properties can improve and significantly improve the accuracy of the model.

In works [22] and [23], the authors use a stress application (called Stressmark) to steal cache from a co-run target application. Stressmark steals cache space and the amount of cache space stolen is determined from an analysis of its known Miss Ratio Curve (MRC). However, a limitation with this approach is that MRC is determined by its interaction with the target applications cache footprint. This behaviour varies during execution and they are only able to determine the average cache-utilization for a given execution. For a target application with varying cache foot print, performance modelling using this approach can be significantly affected. The problems of this approach is mitigated by Cache Pirate [24].

In Cache Pirating [24], the authors present a low-overhead method to measure the performance of an application as a function of cache space availability. Cache Pirating does not require any modification to the existing hardware or the operating system. It is achieved by co-running a pirate application that steals cache space from the target application. The pirate needs to ensure that its working set is always retained in the last level cache, while being tunable. It works in unison with the cache replacement policy and manages it access pattern such that the data of the pirate is always retained in the cache. By tuning the pirate application to steal varying amounts of cache space, they are able to model the performance of the target application.

Fairness via Source Throttling (FST) [25] and Per-thread cycle accounting (PTCA) [26] estimate slowdown due to both shared cache and main memory interference. They determine the effect of interference on slowdown at a per-request granularity. Specifically, to estimate an application's isolated execution time, both FST and PTCA determine the number of cycles by which each request of the application is delayed due to interference at the shared cache and main memory. The drawback of this approach is that with the abundant amounts of parallelism in the memory subsystem, the service

of different requests will likely overlap. As a result, estimating the effect of interference on slowdown at an individual request granularity is difficult and leads to inaccurate estimates for FST and PTCA.

Application Slowdown Model (ASM) [27] is a new technique that accurately estimates application slowdowns due to interference at both the shared cache and main memory without a priori application knowledge. It is based on the observation that an applications performance is strongly correlated to the rate at which the applications accesses the shared cache. The slowdown is then determined dynamically by two steps: First, by estimating the average cache-miss service time had it been run alone, and second, by estimating the number of cache-misses that would have been hits if the application did not share the cache with other applications. The first estimation is achieved by prioritising the memory requests of the application for short time periods and the second estimation by using an auxiliary tag store. ASM works better than FST and PTCA as it does not consider per-request effect and instead takes into account aggregated behaviour.

2.2.4 Scheduling

In [28] the application is profiled statically to predict interference and identify safe co-locations for VMs. It mainly focuses on predicting which applications can be co-run with a given application without degrading its QoS beyond a certain threshold. The limitation of static profiling introduces a lack of ability to adapt to changes in application dynamic behaviour. Paragon [29] tries to overcome the problem of complete static profiling by profiling only a part of the application and relies on a recommendation system, based on the knowledge of previous execution, to identify the best placement for applications with respect to interference. Since only a part of the application is profiled, dynamic behaviours such as phase changes and workload changes are not captured and can lead to a suboptimal schedule resulting in severe performance degradation. Quasar [30], extends Paragon and takes an alternative approach to cluster management that does not require users to specify cluster requirements in the form of resource reservations but instead takes it in the form of performance constraints. Quasar is then capable of determining the right amount of resources required to meet the constraints. Bubble-Flex [31] is a runtime method for mitigating performance interference by phasing in and phasing out

batch applications upon detecting QoS violations. It predicts QoS violations by generating load in the memory subsystem in bursts during runtime to generate a sensitivity curve. Their approach introduces additional stress on the memory subsystem, which can itself cause interference and batch applications are unable to fully exploit periods of low utilization. Interference Management for Distributed Parallel Applications in Consolidated Clusters [32] studies the impact of local interference on the execution of distributed applications. The major difference for a distributed application is that interference on a local node can affect the end to end performance of the application spanning many nodes. The authors characterize the effect of interference for various distributed applications with varying intensities and distribution of interference. Based on this characterization, the authors design a interference propagation model that estimates how interference on a subset of nodes affects the end-to-end performance. Using the proposed method, they develop an interference-aware placement based on simulated annealing, which can efficiently consolidate multiple distributed applications

Tracon [33] utilizes modeling and control techniques from statistical machine learning to achieve interference aware scheduling. Essentially, it consists of an interference prediction model that infers application performance from resource consumption observed from different VMs and adapts the model at runtime for efficient resource management.

2.2.5 Detecting Contention

Several detection techniques have been proposed to identify contention. Most works [34], [35] use the hosts' hardware performance counters to detect interference. LLC miss rate is widely used to detect contention [12]. CPI2 [36] rely on cycles per instruction and look for significant statistical deviation to detect interference. It relies on the existence of thousands of tasks for a job to find statistical outliers from normal behaviour. Hardware performance counters and operating system metrics need access to host information and as such is more suited as a cloud provider solution. Mukerjee et al. [37] uses a software probe that executes sections of micro-benchmark code specifically designed to detect contention on the memory hierarchy. The execution time of the probe is then compared against the execution time during an isolated run and any statistical difference is used to flag interference.

Yasaman et al. [38] propose a machine learning based interference detection technique. It specifically uses collaborative filtering to predict whether a given transaction is adversely suffering from interference and does not require explicit authoring of models. The major difference from the remaining work is that they don't rely on hardware performance counters and as such is applicable for the cloud subscriber to detect contention.

Casale et al. [39] also propose a subscriber centric solution to detect contention on processor cache. It continuously monitors the execution times of a set of benchmarks which is then compared with isolated execution of the benchmark and CPU steal metric to predict whether or not a VM is affected by interference.

Chapter 3

Contributions

In this chapter, the main research questions and the challenges are presented. Each set of questions is then discussed and the related contributions are summarized according to the approach we chose. The specific contributions of this thesis are presented in detail in chapters 4, 5 and 6.

The research aims to address questions from different perspectives, which incrementally add up to build a solution that is capable of efficient resource management that is performance aware while providing high utilization. Achieving this entails understanding the effects of performance interference and we approach it from 2 perspectives:

3.1 Performance Interference from a single host perspective

Single host perspective deals with performance degradation that manifest purely from contention in shared resources and does not consider scenarios where the application workload overloads the hosts capacity. Difficulties to model and predict performance interference has resulted in the heavy handed approach of disallowing any co-locations with performance sensitive applications, a major contributor to low machine utilization in data centers [3]. Recent work has seen proposals to predict interference and minimise QoS degradations by relying on static approaches based on prior profiling of applications [40, 28]. Even with an ideal profiling technique, it is impossible to fully characterize an application before run-time in order to prevent interference

and improve utilisation. This is because applications can have varying and sometimes unpredictable inputs/workloads during run-time and services may run for extended periods making it infeasible to profile. Addressing these challenges to mitigate performance interference requires analysis during the run-time to deal with long running jobs and varying workloads. Thus, assuring QoS for sensitive applications while allowing co-location continues to remain a challenging problem. The specific challenges that exacerbates the difficulty in providing performance guarantees while ensuring high utilization stems from the following reasons:

- Unknown Application: Application never seen before, hence no information about the application behaviour is available. Consequently, the optimal performance is unknown.
- Dynamic Workload: Varying and sometimes unpredictable workload voids any kind of offline characterization.
- Unknown Expiration: Web applications for example run for extended periods. Such applications cannot be fully characterized offline.
- Quantifying Degradation: Predicting and quantifying the degradation experienced from contention is application specific and difficult to model.

Q1: Is it possible to improve machine utilization without violating SLO of performance-sensitive services using a black-box approach? What are the right dynamic configuration techniques to achieve the same? What kind of prediction helps minimise SLO violations in a co-located setting?

The challenges outlined in the previous section motivate the need for a solution that is dynamic and run-time aware in order to improve utilization without sacrificing the QoS requirement of latency critical services. We design Stay-Away, a generic and adaptive mechanism capable of performing a run-time analysis to execute best-effort batch applications without sacrificing the QoS of latency sensitive applications. Applications typically don't use all the requested resources during the life cycle of their execution because of changes in workload intensity and inherent phase changes. A phase change is defined as a change in the major share of resource consumed by an application. For example, an application can be mostly CPU intensive for a certain period

and be I/O intensive at other times. As a result, not all the resources are used at all times. Apart from these phase changes, varying workloads often result in periods of low utilisation. Stay-Away leverages on these periods of low workload intensity and phase changes to improve utilisation by executing best-effort batch applications without sacrificing the QoS of latency sensitive applications.

To summarize, the specific contributions in this chapter are:

- We present the design of Stay-Away, a generic and adaptive mechanism to improve machine utilization by executing best-effort batch applications alongside latency sensitive applications while providing performance guarantees. Stay-away learns QoS violations when application execute together and progressively learns a model to avoid the learned QoS violations by throttling batch applications.
- We design a methodology to model real-time transitions of the VM states to be aware of dynamic changes across the environment in order to prevent known QoS violations before they occur.

3.2 Performance Interference from a distributed system perspective

In this section, we focus on scenarios where an application workload can overload a hosts capacity and necessitates the need for elastic scaling to provide performance guarantees. Under such circumstances, we show that performance interference introduces additional challenges that impact the accuracy of elastic scaling. Although the eventual impact of performance interference is always performance degradation, the performance degradation manifests from not accounting for the role of contention. Huang et al. [41] show that contention in one shared resource has a cascading effect on other resources and results in performance consequences that may not be easy to model. Similarly, we explore the implications of performance interference that go beyond performance degradation. With Elastic scaling as a use case, we demonstrate that not accounting for performance interference can affect the reliability of decision making phase and the accuracy of the actuation phase. As a result, not only does it degrade the performance but it can also result in

undue resource provisioning subsequently increasing the cost. We provide an overview of elastic resource provisioning and then present the contributions.

Elastic resource provisioning is used to guarantee service level objective (SLO) with reduced cost in a Cloud platform. Elastic scaling typically consists of an elasticity controller that allows to meet the demands of the changing workload by adding or removing resources when required. It consists of a *decision making* phase that decides when to scale out/down and an *actuation* phase that acts as an interface to resource infrastructure and adds/removes resources. We show that performance interference affects both the decision-making phase and the actuation phase. Existing elasticity controllers are either unaware of performance interference or over-provision resources to meet the SLO requirements. In this thesis, we take a holistic view on elastic scaling from a multi-tenant perspective and show that performance interference affects both service reliability and accuracy.

3.2.1 Service Reliability (Decision Making):

Specifically, we try to answer the following research questions:

Q2: What are the consequences of performance interference apart from degradation in performance? Do these consequences have any implication on the reliability of metrics used for elastic scaling? Is it possible to circumvent/mitigate the negative implications?

CPU utilization [42, 43, 44, 45, 46] and workload intensity [47, 48, 49, 50, 51] are two widely used indirect metrics in the decision-making phase to decide when to scale a system. They are widely used as they are easily available without any significant overhead and correlate well with the measure of service quality such as latency. For example, high CPU utilization corresponds to high latency and low CPU utilization corresponds to low latency. An elasticity controller that relies on CPU utilization learns a model that captures this correlation which is then used in the decision making phase to decide when to scale. These metrics work well in an isolated environment and are widely used. We explore the reliability of these indirect metrics in a multi-tenant environment and find that these metrics become unreliable when performance

interference is unaccounted. Our results show that performance interference skews their correlation with latency thereby resulting in inaccurate decisions.

Our main contribution is Hubbub-Scale; an elasticity controller that achieves predictable performance in the face of resource contention without any significant overhead. We facilitate this by designing a middleware that provides an API to quantify the amount of pressure the co-running VMs put on the target system. Specifically our contributions are:

- We show that OS configuration, performance interference and power-saving optimisations stand in the way of predictable performance and impact the decision making phase. While OS configuration and power-saving optimisations can be controlled, performance interference is inevitable in a multi-tenant system and needs to be modelled.
- In the presence of performance interference, indirect metrics used for elastic scaling cease to accurately reflect the measure of service quality, consequently affecting the scaling accuracy and reliability of indirect metrics.
- We build Hubbub-scale, an elasticity controller that is reliable in the presence of performance interference and achieves high resource utilization without violating the SLO.

3.2.2 Service Accuracy (Actuation):

Q3: Does performance interference impact the accuracy of elastic scaling? Is it possible to augment existing elasticity controllers to be aware of performance interference? Does augmentation reduce SLO violations and improve the scaling accuracy? Can the augmentation also improve the system utilization?

Here, we focus on the actuation phase of the elasticity controller that acts as an interface to the resource infrastructure. We show that when elasticity controllers are unaware of interference during the actuation phase, it either results in long periods of unmet capacity that manifest as Service Level Objective (SLO) violations or results in higher costs from over provisioning. We augment the elasticity controller to be aware of interference and significantly reduce SLO violations and save provisioning costs. We consider Memcached

for elastic scaling, as it is widely used as a caching layer, and present a practical solution to augment existing elasticity controllers in 3 ways: (i) At the ingress point by load balancer reconfiguration (ii) by reducing the convergence time when scaling out and (iii) by taking informed decisions when scaling down/removing instances. We achieve this with the help of hardware performance counters to quantify the intensity of interference on the host. This is achieved with the help of a middleware that exposes an API for VMs to query the amount of interference in the host. Decisions by the elasticity controller is then augmented with the help of this information. Our main contributions are:

1. We show that the maximum workload any VM can serve within a SLO constraint is severely impacted by interference. An immediate consequence of this impact is an increase in the time taken for the scaling out process to converge which results in increased SLO violations. We also show that this resulting period of SLO violation is directly proportional to the time taken to spawn and prepare VMs.

2. We design and develop a solution to augment elasticity controllers to be aware of interference. Our solution quantifies the capacity of a VM based on the interference experienced on the host by modelling the performance of the target application. With this we are able to reduce the impact of interference on SLO violations by reconfiguring the load balancer, reducing the convergence time when scaling out and by removing highly interfered instances from the cluster when scaling down.

3. We perform experiments with Memcached and compare our solution against a baseline elasticity controller that is unaware of performance interference. We find that with augmentation we can not only reduce SLO violations significantly but also save provisioning costs compared to an interference oblivious controller.

Chapter 4

Stay-Away: Protecting Sensitive Applications from Performance Interference

Abstract

While co-locating virtual machines improves utilization in resource shared environments, the resulting performance interference between VMs is difficult to model or predict. QoS of sensitive applications can suffer from resource co-location with other resource intensive applications. The common practice of overprovisioning resources helps to avoid performance interference and guarantee QoS but leads to low machine utilization. Recent work that rely on static approaches suffer from practical limitations due to assumptions such as a-priori knowledge of application behaviour and workload. To address these limitations, we present Stay-Away, a generic and adaptive mechanism to mitigate the detrimental effects of performance interference on sensitive applications when co-located with best-effort applications. Our mechanism complements the allocation decisions of resource schedulers by continuously learning the favourable and unfavourable states of co-execution and mapping them to a state-space representation. Trajectories in this representation are used to predict and prevent any transition towards performance degradation of sensitive applications by proactively throttling the execution of batch applications. Experimental results with realistic applications show that it is possible to guarantee a high level of QoS for latency sensitive applications while also improving machine utilization.

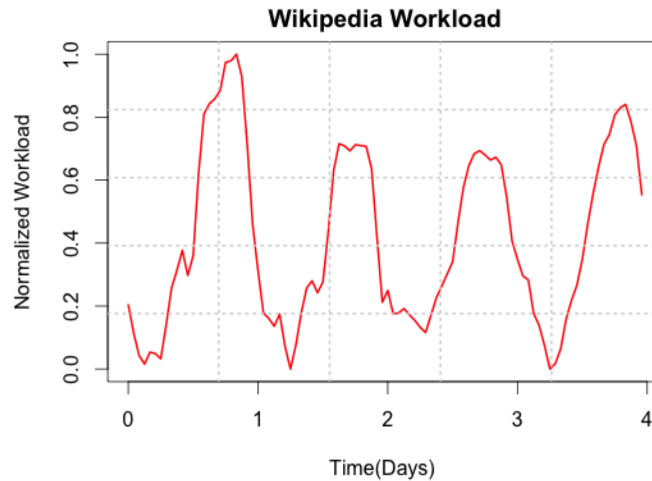


Figure 4.1: Total Workload variation of Wikipedia during the period 1/1/2011 to 5/1/2011

4.1 Introduction

In this chapter, we present Stay-Away, a generic and adaptive mechanism capable of performing a runtime analysis to execute best-effort batch applications co-located with latency sensitive applications without sacrificing the QoS of latency sensitive applications. Applications typically don't use all the requested resources during the life cycle of their execution because of changes in workload intensity and inherent phase changes. A phase change is defined as a change in the major share of resource consumed by an application. For example, An application can be mostly CPU intensive for a certain period and be I/O intensive at other times. As a result, not all the resources are used at all times. Apart from these phase changes, varying workloads often result in periods of low utilisation. Figure 4.1 shows the total read workload for Wikipedia obtained from trace [52]. The workload follows a diurnal pattern with clear periods of low workload intensity. Stay-Away leverages on these periods on low workload intensity and phase changes to improve utilisation by executing best-effort batch applications without sacrificing the QoS of latency sensitive applications.

Stay-Away periodically monitors the resource usage metrics of every Virtual Machine in the host, yielding a time series of measurement vectors. These vectors are then mapped onto a two dimensional space such that similar mea-

surement vectors group together. QoS violations manifest during executions when VMs contend for a resource. This results in measurement vectors that deviate significantly from the measurement vectors of normal executions. As a result measurement vectors of QoS violations are mapped farther away from the group of normal executions. Once this mapping is done, Stay-Away then predicts any progression towards a QoS violation by performing a continuous spatial and temporal analysis of the two dimensional space to identify transitions, their rate and direction. Upon detection of any transition towards a QoS violation, Stay-Away throttles the batch application to avoid a contention before it occurs.

To summarize, the specific contributions in this work are:

- We present the design of Stay-Away, generic and adaptive mechanism to mitigate the detrimental effects of performance interference on sensitive applications when co-located with batch applications.
- We design a methodology to model real-time transitions of the VM states to be aware of dynamic changes across the environment in order to prevent known QoS violations before they occur.
- Additionally, we discuss how this methodology can serve as a template for repeatable experiments.

We experiment with VLC streaming server and a Webservice, co-located with different set of batch applications. Our results indicate that using Stay-Away, we are able to guarantee a high level of QoS, and are able to increase the machine untilization by 10%-70%, depending on the type of co-located batch application.

4.2 Background

In this section, we highlight the differences between scheduling and dynamic reconfiguration, and provide the necessary background for understanding Multi Dimensional Scaling (MDS), a key component of our approach.

4.2.1 VM Placement

The problem of mitigating inter-VM interference can be seen from two different perspectives: VM placement and dynamic reconfiguration. The VMs can be scheduled to co-locate in a manner such that they minimize performance interference between each other. For instance, co-locating a memory intensive application with a CPU intensive application is much better than co-locating 2 memory intensive applications together. Papers like Bubble-up [28] and Paragon [29] solve the problem of interference by deciding which VMs to co-locate. In other words, these systems leverage the freedom to co-locate VMs such that there is minimal interference. Their techniques rely on application characterization and fails to address the challenges of dynamic workload and long running applications.

Alternatively, interference can be alleviated even after VMs are co-located. This can be achieved through dynamic reconfiguration. Dynamic reconfiguration may include resource scaling (dynamically increasing the amount of resources allocated to a VM), VM migration or relaxing the guarantees on some VMs. For example, a VM might be sharing a socket with other VMs leading to an interference at cache level, but by dynamically reconfiguring the VM to use the entire socket in an isolated way, the processing power increases and also VMs do not interfere at cache level anymore. This is possible only if additional resources are available for scaling. Cost of dynamic reconfiguration is an important concern. An adaptation is feasible only If the benefits accrued by reconfiguration outweighs the cost involved. VM migration is slow and involves a high cost.Yet another dynamic reconfiguration technique is to throttle the VMs that cause interference and don't need strict guarantees with performance. This does not incur a high cost and is instantaneous.

For these reasons, Stay-Away relies on throttling VMs when resource contentions are about to happen. This affects the performance of the throttled VMs and require them to be best-effort. We introduce the constraint that only best-effort batch applications can be scheduled with latency sensitive applications. With this constraint in place, we are able to achieve a high level of QoS and improved utilisation. Stay-Away is not a scheduler. It relies on dynamic reconfiguration and can complement from schedulers like Choosy[53] that allows scheduling with constraints.

Violation of QoS can happen because of 2 reasons: the system being unable to meet the demands because of unusual or extremely high workload overloading the host or because the system is unable to meet the demands because of interference from co-located VMs. In this chapter, we assume that any violation is only because of interference, and workloads overloading hosts that require scaling are discussed in chapter 5 and 6

4.2.2 Multi-Dimensional Scaling

Multi-Dimensional Scaling (MDS)[54] has the property of being able to represent a high dimensional space $\mathbf{R^m}$ in a lower dimensional one, for instance $\mathbf{R^2}$, preserving the relative distances, i.e. the absolute value of the distances is lost, but the points are rearranged in a 2D space so that the relative distances between points in the plane correspond to the relative distances in the high dimensional space. Each object or event is represented by a point in a 2D space. The points are arranged in this space so that the distances between pairs of points have the strongest possible relation to the similarities among the pairs of objects. That is, two similar objects are represented by two points that are close together, and two dissimilar objects are represented by two points that are far apart. Unlike a projection operator such as PCA [55] or a manifold discovery algorithm [56], which gives superposition in the direction of projection, MDS creates a new representation based on the distances between points

The algorithm for assigning the points in a lower dimensional space preserving the relative distances between points is based on the idea of stress majorization, which assigns the coordinates by minimizing a loss function based on the weighted sum of the differences between the euclidean distances on the original space and the distances on the representation plane. The loss function is defined as $Loss(X) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (Dist(x_i, x_j) - \delta_{i,j})^2$, where $x_i, x_j \in \mathbf{R^m}$, the matrix X consist of the concatenation of the state vectors x_i and $\delta_{i,j}$ corresponds to the relative distance of the represented points i, j on the plane $\mathbf{R^2}$. The loss function can be minimized by using Scaling by majorizing a convex function (SMACOF) algorithm, which minimizes a quadratic form iteratively.

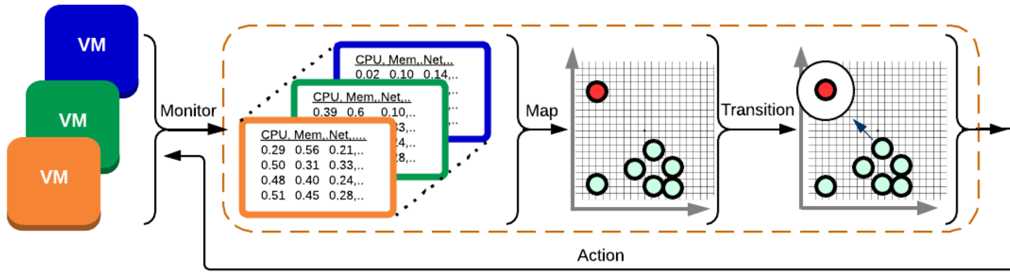


Figure 4.2: Stay-Away mechanism. This figure illustrates the 3 steps: Mapping, Predicting Transition and Action. The darker circle on the top of the map represents a QoS violation.

4.3 Stay-Away Mechanism

The key insight of Stay-Away is that, the closer the resource usage resembles a contention that was previously responsible for QoS degradation, the more the sensitive application progresses towards a QoS violation. The Stay-Away runtime is a middleware between the VMs and the underlying resource, and runs on each host periodically following a three step mechanism: Mapping, Prediction and Action as shown in figure 4.2. All the three steps are performed in each period.

4.3.1 Mapping

Stay-Away employs a set of continuous VM-state resource usage snapshots to capture patterns of VM behaviour during application runs. Specifically, the runtime-learning phase begins by periodically measuring a set of VM metrics such as CPU, memory, I/O, network traffic for all VMs in the physical host as a vector of measurements M(t) = <VM_i-CPU, VM_i-Memory, VM_i-I/O, VM_i-network> for all VMs at time t. Stay-Away does not impose any limitation on the choice of metrics to be used. Ideally, the right metrics to use are those that characterize the load on the resource subsystem we are interested in. For example, the performance counter bus transactions for each VM can be used to characterize the load on the memory bus. These measurement vectors are then mapped into a lower dimension using MDS. This maintains

the topological properties (relative distances) of the high dimensional space and similar measurement vectors remain close to each other, while dissimilar measurement vectors are placed farther apart. The mapped measurement vector in the lower dimensional space is called a *mapped-state*.

The mapping of each measurement vector over time captures the temporal behaviour of the execution. The path traced by the mapped-state is the trajectory of execution. Stay-Away relies on the application to report whenever a QoS violation happens in order to label the mapped state corresponding to the QoS violation. Alternatively, using IPC to detect QoS violation is explored in other works[31]. The *mapped-state* corresponding to a QoS violation is called a *violation-state*.

The measurement vectors can be mapped on to any lower dimension, we specifically selected a 2D representation for the following reasons:

- *Interpretability*: As MDS preserves the relative distances between measurement vectors, it helps understand the patterns and behaviours during the temporal evolution of the co-located VM execution. The metric preservation property of MDS also maintains the angles and the directions of the trajectories, which means that a prediction in a plane (y, x), gives a reliable representation of the behaviour on the high dimensional space.

- *Parameter Estimation*: A natural technique for forecasting in high dimensions is Vector Autoregressive Models (VAR)[55]. In high dimensional spaces, the number of samples needed for a reliable estimation of parameters by means of histograms (explained in section 4.3.2.3) increases exponentially with the dimensionality and also the domain of values for the parameters increases with increasing dimensions, leading to unreliable parameter estimation. A 2D representation of the trajectories gives prediction models with two parameters, which can be estimated reliably from a small sample.

4.3.2 Prediction

The second step of Stay-Away is to predict the future state of the execution based on the observations so far. The mapping phase produces a two-dimensional map which is then used by the predictor to forecast the transition

of the execution behaviour. Specifically, we are interested only in knowing if the execution progresses towards a *violation-state*. Once it is fairly certain that a QoS violation is likely in the future, Stay-Away can then steer away from QoS violation by throttling the batch application. The predictor has 2 goals:

- To prevent an impending violation
- To allow the system to progressively learn about new violation states

These are conflicting goals, since, letting the applications to execute till a QoS violation happens in order to learn implies that a preventive action should not be taken. This is overly conservative on the batch applications and degrades the performance of sensitive applications. However,throttling the batch application based on incomplete information is overly aggressive on the batch applications and restricts state-space exploration. The predictor needs to strike a balance between the two to consistently anneal to the correct QoS value without being overly aggressive nor overly conservative. In the following subsections, we explain how the predictor strikes a balance between the two and how a future *mapped-state* is estimated.

4.3.2.1 Should a prediction rely only on known QoS violations?

When observing a resource contention causing a QoS violation, the system state is mapped on to the state-space and marked as *violation-state*. During every period, the predictor tries to estimate the position of the future *mapped-state*. If the estimated *mapped-state* overlaps with the *violated-state*, then a QoS violation is likely.

Any *mapped-state* or *violation-state* represent only the observed system behaviour. Since the *violation-states* also correspond to specific measurement vectors, and is marked as a point in the 2D space. If throttling of batch applications is done only based on exact overlap of the estimated *mapped-state* with *violation-state*, it limits the prediction to only seen states of violation. It is highly likely that the nearby neighbouring states around the *violation-state* also correspond to a QoS violation as they are separated only by minor deviations. For example, in a QoS violation corresponding to a particular VM consuming 250MB of memory, minor deviations such as 255MB of memory

usage would still cause a QoS violation even though the exact value of 255MB was not seen. If the predictor can take advantage by extending the range to account for unseen violations, it can prevent impending violations without having to capture the violation explicitly.

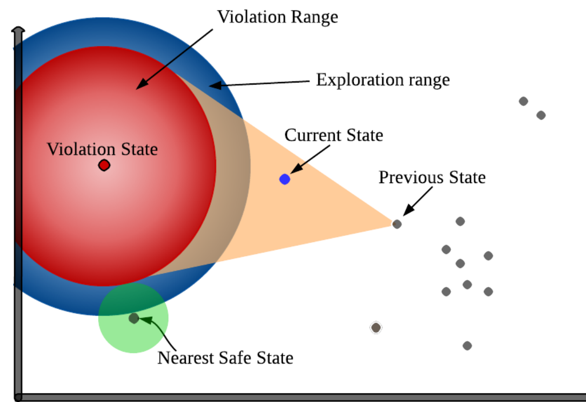


Figure 4.3: State-space exploration

The unexplored neighbourhood area around the *violation-state* is called the *violation-range*, marked as a circle and is shown in figure 4.3. The *violation-range* is an approximation and corresponds to that area in the state space which the system hasn't seen yet but deems as the neighbourhood that would contain *violation-states* if a state were to be mapped in that range. Consequently, if an estimated *mapped-state* falls within a violation range, the batch application is throttled. Thus, a *violation-range* with a big radius would lead to aggresively throttling batch applications and a *violation-range* with a very small radius could lead to multiple QoS violations. In the next subsection, we explain how Stay-Away progressively attains accuracy and strikes a balance.The *exploration-range* is that neighbourhood which the system assumes safe. The predictor relies on a heuristic to predict the *violation-range* and progressively aims to attain accuracy. The area of the *violation-range* depends on the nearest known *safe-state* in the state-space. *Mapped-states* that do not correspond to a violation are called *safe-states*.

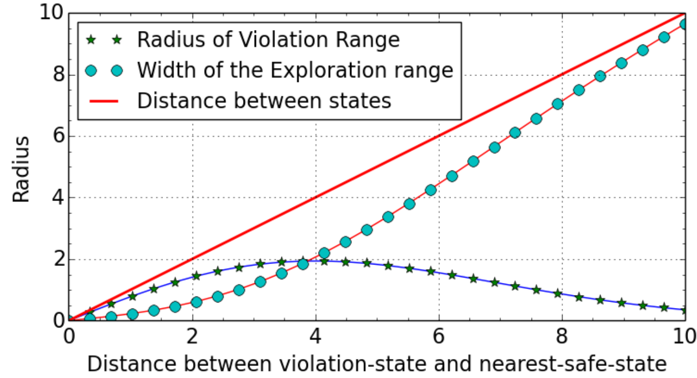


Figure 4.4: Variation of the radius of *violation-range* as distance between the *violation-state* and nearest *safe-state* varies

4.3.2.2 How Far to explore?

The radius of the *violation-range* is modelled as opposing forces (repulsion) between the *violation-state* and the nearest *safe-state*. The choice is intuitive: the closer there is a known *safe-state*, the lesser is the area of the *violation-range*. Initially the range is an approximation and as more states are explored, the representation gets more accurate. The radius of the *violation-range* is defined as the distance between *violation-state* and the nearest *safe-state* scaled by a Rayleigh distribution. It is important not to define the entire distance between a *violation-state* and the nearest *safe-state* as the radius of *violation-range* as it would prevent the system from exploring new states closer to the *violation-range*. A Rayleigh distribution is used to allow for the exploration range to adapt depending on the distance between the nearest *safe-state* and the *violation-state*. Ideally, the size of the exploration range should fade as the distance between the nearest *safe-state* and the *violation-state* gets closer. The radius of the violation range is given by:

$$R = de^{\frac{-d^2}{2c^2}}$$

where d is the distance between the nearest safe state and the violation state. c is the shape factor and is defined as the median of the coordinate range of the mapped space. It follows from the observations that as the distance between these states increase, it is safer to increase the size of the *exploration-range* as we are more likely to be farther away from any unseen *violation-states*. However, as the distance between these states get closer, the exploration range

needs to fade. Figure 4.4 shows the variation of the radius of violation range and the size of exploration range as the distance between these states vary.

4.3.2.3 When to act?

During the period of co-located execution, the system transitions through different states determined by the extent and type of resources being used by the VMs. In order to minimize the effect of performance interference, Stay-Away needs to predict any transition towards a violation and take a preventive action. The state transitions are specific to the applications running on the VM and can be:

- Gradual transitions, marked by resource consumptions such as memory usage where the memory allocated for the application typically varies gradually over time and as a result presents a consistent transition. Transitions are not marked by one application alone but instead by the combination of resource usage from all applications. As a result, the vector measurements in combination may or may not strictly follow a linear movement. However, the rate and pattern in transition becomes more apparent if the co-located application experiences minimal phase transition during its period of execution.
- Instantaneous transitions, marked by resource types such as CPU usage that could vary as instantaneous spikes. These sudden changes contribute to state transition in quick successions reducing the reaction time for any preventive action. For example, consider a *violation-state* characterised by a measurement vector with a contention at the level of CPU. If in future both applications contend for CPU, the transition to violated state is very quick giving almost no time for the system to react.

We have seen that the extent of *safe-states* are defined by the *exploration-range* and Stay-Away tries to avoid entering the *violation-range* to avoid violations. While these ranges act as a demarcation in space, the rate and direction of transition measure the temporal evolution or the progression over time.

State transitions are the result of complicated responses to an applications internal behaviour, and interactions between the co-located applications. It

is not immediately obvious how a continuous movement process should be modelled and parameterized, since movement is multi-dimensional, combining both spatial and a temporal dimension. Because it is impossible to accurately model all of these interactions, they need to be modelled stochastically i.e. with intrinsically random velocities and orientations that can be summarized by well-defined probability densities and associated parameters. However, we observed that the accuracy of the prediction model suffers severely when all the state transitions are modelled using a single model. This is because every application has a characteristic behaviour and sometimes repetitive phases [57]. At any point in time, one of these 4 execution modes hold true:

- No application is running
- Batch application runs alone
- Latency-sensitive application runs alone
- Co-located execution of both batch application and latency-sensitive application

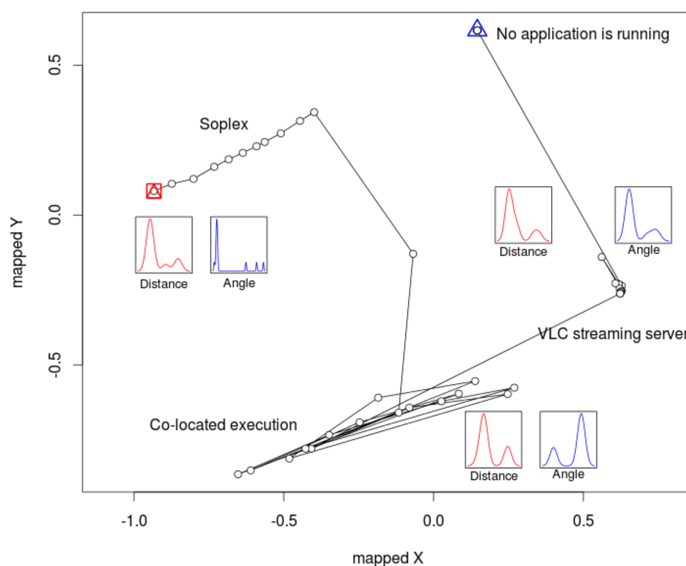


Figure 4.5: All 4 execution modes when VLC streaming is co-located with Soplex from SPEC CPU 2006

Figure 4.5 shows the state transition for an execution lifecycle comprising of VLC streaming server and Soplex from the SPEC CPU 2006 benchmark suite. The state transitions begins with no application running, followed by executing VLC streaming. Shortly after, the batch application is scheduled to execute and the states transition to co-located execution. Finally, VLC streaming finishes its execution and the batch application executes in an isolated fashion until it finishes. We can clearly see that each execution mode forms clusters and has a different pattern for trajectory. While VLC streaming is characterised by short bursts of correlated movement, Soplex follows a linear trajectory with a consistent orientation and slightly varying step length. Co-located execution on the other hand experiences an oscillating trajectory with bigger step lengths. As a result, modelling all the different execution modes using a single model fails to capture the inherent patterns and sequence specific to each execution mode. The trajectory pattern experienced in each of these execution modes is different. We experimented with numerous other co-locations of applications and observe that the life cycle of an execution steps through different modes and the trajectory pattern has a high dependence on the current execution mode. Our prediction model stems from this observation and as such no single prediction model can accurately model all the state transitions. The state transitions are broadly categorized into these 4 distinct execution modes, each with a different prediction model. Since Stay-Away runtime is a middleware managing the VMs, it can any time determine the current execution mode the system is in.

Marsh et al. [58] noted that a good description of the trajectory is achieved when the measured parameters and the relationships between them are sufficient to reconstruct characteristic tracks without loosing any of their significant properties while relying on a minimum set of relatively easily measured parameters. Based on a literature review [59], we have noticed that the trajectory can be fairly accurately modelled by the following parameters:

- Distance: The distance d between successive positions
- Absolute angle: The absolute angle α_i between the x direction and the step built by transitions from positions i and i+1

No static trajectory model can be assumed even within an execution mode for different sets of co-locations as each application has a different characteristic

behaviour of its own. For a particular combination of batch application and latency sensitive application, co-located execution mode may show characteristics of a Biased Random Walk [60] whereas for a different combination, the execution mode may follow the trajectory model of levy flight [61]. Levy flight trajectories were observed for applications that experiences sudden phase changes. Because of these differences, we cannot rely on a static model and have to learn the behaviour during the execution.

To characterize the trajectories, we capture the behaviour of each execution mode by the probability density function (*pdf*) of the parameters: distance d and absolute angle α_i. The underlying measurement is a histogram. In Figure 4.5, we plot the smoothed version of the histogram using kernel density estimation and show the corresponding *pdf* of both distance and angles for different execution modes. The skew in the distribution indicates that the trajectory is biased and not random (with equal probabilities for all step lengths and angles). The bias indicates that the likelihood of certain step lengths and angles are higher than the others and this helps model the prediction with high accuracy. We experimented with many different set of applications and co-locations and always observe a bias in the trajectory. Therefore, after a few observations have been made, a first approximation of the *pdf*s for both parameters can be derived. A random set of samples are then generated following the histogram using the inverse transform method, which computes a mapping from a uniform distribution to an arbitrary distribution (i.e. the distribution from the histogram) [62]. This allows us to predict a set of new states around the current state and models the uncertainty in the likely position of the future state. Because of the inherent behavioural patterns in applications that cause a bias, with 5 samples to model uncertainty, we are able to achieve more than 90% accuracy on average for all the different co-locations we experimented with in section 4.7. Once the uncertainty is modelled, the generated states are examined to see if they fall within a violation range. Whenever a majority of the generated sample set fall within a violation range, Stay-Away takes an action to prevent degradation.

4.3.3 What Action to take and When to Stop?

To throttle the execution of the batch application, Stay-Away sends a SIGSTOP signal to pause the batch application and SIGCONT to resume its execution.

Once paused, the system does not resume the batch application until the system believes that resuming the batch application will not cause a performance degradation for the sensitive application.This belief is based on the distance between the consecutive states of isolated execution of the sensitive application. Upon throttling, the system moves to a different execution mode. Only the sensitive application executes and prediction model is adapted to the new execution mode. Note that, it is impossible to have a violation in this execution mode as there is no interference. This is also conformed in the state-space representation. If the performance-sensitive application continues to remain in the same phase or continues with the same workload after the batch application is paused, the states that follow roughly map to the same vicinity in the 2D space. An increase in the distance indicates a phase change or change in workload intensity of the sensitive application and is likely to have transitioned from contending for the bottleneck resource. Stay-Away has a learning parameter β, which is the maximum allowed distance between the states before resuming the batch application. Initially β is set to 0.01. Once the distance exceeds β, the system resumes the batch application. However, if resuming the batch application immediately leads to a violation, it indicates that the phase change of the performance-sensitive application was not enough to avoid degradation and the system increments β by a small amount. Over time, β attains accuracy. It is possible that the sensitive application does not experience any phase transition and in such scenarios, the batch application would starve indefinitely. To account for this, Stay-Away uses a random factor to resume the execution of the batch application when the distance falls below β for a long time. This is done in hope that the batch application may experience a phase transition and as such avoid degradation. However, if the batch application continues to degrade performance of the sensitive application, it is paused again.

4.4 Optimisations and Overhead

In order to achieve efficiency, we need to address a few pre-processing problems before feeding the measurement vectors to MDS. Depending on the metrics chosen, the range of values for each metric may vary significantly. For example, while CPU usage ranges between 0 and 100, memory usage does not have a fixed upper limit as each VM could be assigned different amounts of memory. This variation causes higher values to introduce a bias that can affect the

accuracy of MDS mapping. The problem is overcome by normalizing all the metric values between [0,1]. Normalisation also helps to cluster metrics with slight variations in the form of noise around the same neighbourhood forming visible clusters.

The SMACOF algorithm used to represent the high dimensional state to a lower dimension solves a quadratic form iteratively and can become computationally expensive as the number of samples increase. The cost of the algorithm is quadratic and we significantly reduce this overhead by choosing one representative sample from the set of samples that are very close to each other (Eucledian distance) and discarding other similar samples. We noticed that this optimization significantly reduces the computation time as it reduces the size of the observation matrix, while preserving the relative position of the different states, the temporal trajectories followed by the evolution of the execution, and their relative position with respect to the violation state. Alternatively, there is existing work in the literature that is capable of doing incremental MDS with high performance and very low overhead[63, 64]. The induced overhead by Stay-Away in terms of resource consumption is very minimal and corresponds to an average 2% CPU usage and negligible memory consumption.

4.5 Scalability

When the number of dimensions increase, finding an optimal configuration of points in 2-dimensional space can become difficult. The best possible configuration in two dimensions may be a poor, highly distorted, representation of the data. This distortion will be reflected in a high stress value. When this happens, the only possible way to find an optimal configuration is to increase the number of dimensions in the mapped space. In our experiments, we found that the representation in a 2-dimensional space is always optimal with low stress value when there are 2 co-locations of VMs. It is not feasible to assume no more than one batch application for co-location. This can, however, be easily circumvented by considering all the batch applications as one logical VM. The monitored metrics of all the batch application are aggregated together to model their collective behaviour as a single logical VM. Since resources are shared between all the batch applications, contention can be accurately represented by a linear composition of resource usage values. However, identifying the

specific batch application responsible for contention becomes difficult by considering their collective behaviour. Apart from the difficulty and the computation involved in identifying the specific interfering batch application, it is also possible that a single batch application alone does not cause a QoS degradation, but a set of batch applications cause a contention. While one batch application may cause a contention for CPU, another application may contend at the level of memory subsystem. The overhead involved in identifying the specific batch applications responsible for contention exceeds the benefits gained. For this reason, upon detecting a transition towards QoS violation, all the batch applications are collectively throttled. Alternatively, groups of batch applications can be iteratively throttled till QoS is guaranteed and each violation state labelled to identify groups of batch application responsible for that specific type of violation. In our current implementation, we collectively throttle all the batch application.

4.6 Template Properties

In case of repeatable latency sensitive applications, the *violation-states* in the generated map from a previous execution can be used as a starting point and is a valid map for a new execution with a different batch application. The state representation for a performance sensitive application is independent of the specific batch applications running on the co-located virtual machines. Although the generated map from the co-located execution depends on the specific batch application that was co-located, the *mapped-states* themselves are representative of load at the resource level. As a result, the captured states for a performance sensitive application doubles as a template that can be used for future executions alongside a different set of application co-locations. For example, consider a latency sensitive application (L) co-located with a batch application(B_A). Their co-located execution generates a map (map-A) with safe states and violated states. The *violated-states* represent QoS violations. If we execute the same latency sensitive application (L) with another batch application (B_B), the *violated-states* from map-A would still correspond to a valid *violation-state* for the new execution. The batch application (B_B) may never map a state in that *violation-state*, but if the co-located execution were to map a state, it will be a *violation-state*. However, it is also important to note that this is highly dependent on the metrics used to capture the state

Workload Name	Combination
Batch-1	Twitter-Analysis+Soplex
Batch-2	Twitter-Analysis+MemoryBomb

Table 4.1: Combination of Batch Applications

space transition and a bad choice can affect the accuracy of the template properties.

4.7 Evaluation

We conducted our experiments with our Stay-Away prototype on a 3.2 GHz dual-socket Intel Core i5 CPU with 4 cores. Each core has a 32KB L1 private data cache, a 32KB L1 private instruction cache, a 256 KB L2 cache and a shared 4 MB L3 cache. The OS is Ubuntu with GNU/Linux kernel version 3.5.0-22.

4.7.1 Experimental Setup

We chose Linux Containers (LXC)[65] because of its ability to provide near-native performance for applications even though it is highly susceptible to performance interference [66]. To evaluate Stay-Away, we use two different types of latency-sensitive applications and show that LXC combined with Stay-Away can achieve a high degree of QoS while improving machine utilisation. We conducted experiments using the VLC[67] media player for video streaming and a Webservice with a memory intensive, CPU intensive and a mix of both CPU and memory intensive workload as the performance sensitive application. Soplex from SPEC CPU 2006 benchmark [68], Twitter influence ranking from the Cloud Suite benchmark [69], CPUBomb from the isolation benchmark suite[70], VLC transcoding and a custom synthetic application that stresses the memory(Memory Bomb) were used as batch applications. Memory Bomb generates stress on the memory subsystem by allocating large chunks of memory and occasionally reading the allocated content. In order to evaluate the QoS and utilization with more than one co-location of a batch application, we setup two different combinations of batch applications shown in table 4.1. Each of the batch application were executed in a different LXC container. We instrumented the source code of VLC 2.0.5 to capture performance metrics

when using VLC to stream a movie in real time to clients. The minimum transcoding rate required to provide real time viewing without any loss of frames at the server side is defined as the QoS threshold.

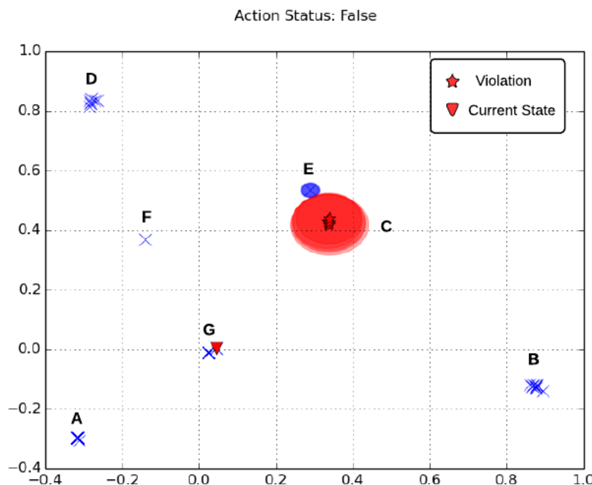


Figure 4.6: Snapshot of instantaneous transition of states when VLC transcoding is co-located with CPUBomb in the mapped space. Action status:False indicates that Stay-Away was not throttling the batch application during the snapshot

The Webservice is setup for analysing and serving data. It consists of a Memcached layer for in-memory data storage and performs analytics, if necessary, before serving the data. The data used for storage and analysis is the open dataset available from [71] and contains periodic network topology information and monitored host metrics of more than 80 nodes which are a part of the community-lab testbed [2]. The Webservice is capable of performing statistical analysis and aggregation of data for each monitored metric and to serve requested data for any specific period. The workload comprises of CPU intensive, Memory intensive and mix of CPU and memory intensive operations.

We begin by illustrating the state-space representation during different execution modes. To illustrate both instantaneous and gradual transitions, we first run two batch applications: transcoding a video with VLC in one LXC container and CPUBomb in another. We chose this co-location to simplify the illustration as both batch applications experience minimal phase transitions

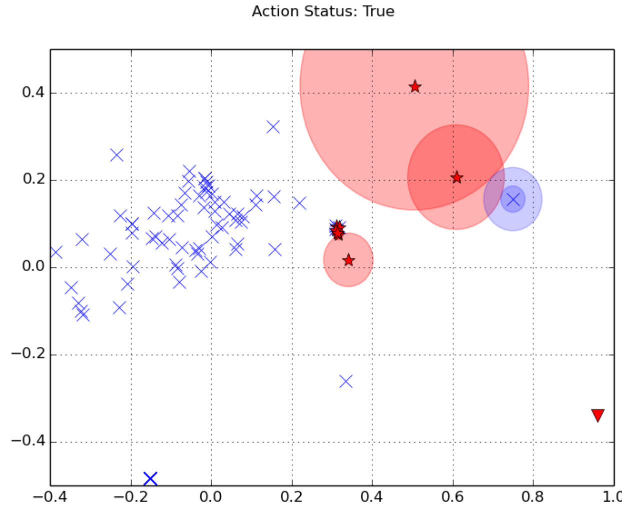
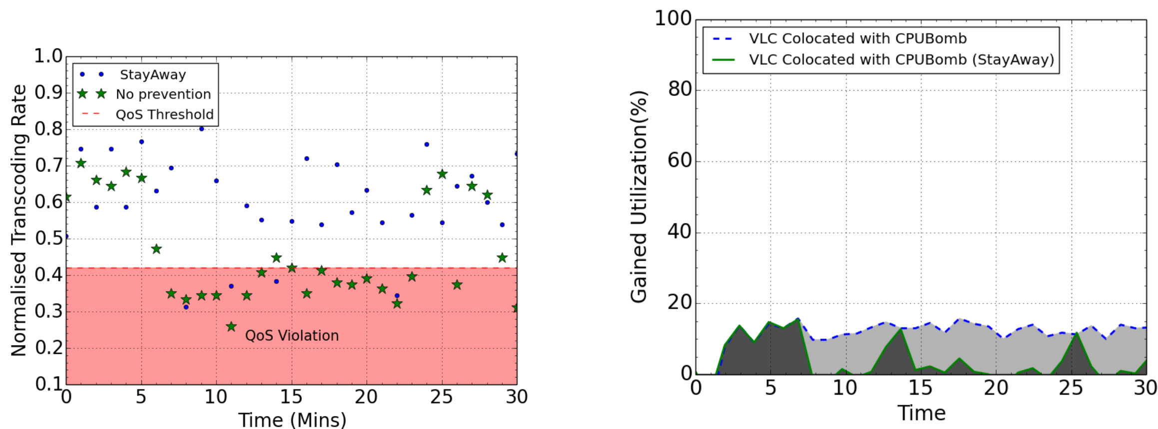


Figure 4.7: Snapshot of gradual transition of states when VLC streaming is co-located with Twitter-Analysis in the mapped space. Action status:True indicates that the batch application was being throttled during the snapshot

during isolated execution. In this contrived, yet representative example for illustrating state space transitions, a violation is said to have occurred when the rate of transcoding frames fall below a certain threshold. Figure4.6 shows a snapshot of the different states the system experiences during the execution life cycle. Darker points represents states with minor transitions that maps closer to each other. In figure 4.6, A corresponds to the state when only CPUBomb was executing. B corresponds to the state when VLC-transcoding runs alongside CPUBomb and C represents the violation state. When Stay-Away takes an action to prevent violation, the corresponding state experienced is represented by D. States E,F,G represent the states during the period of transition. Figure 4.7 shows the gradual transition observed when VLC was used as a streaming server (performance sensitive) and co-located with the Twitter-Analysis (batch).

4.7.2 QoS and Utilization

We first evaluate the effectiveness of Stay-Away for enforcing targeted QoS without any prior profiling of the application. Figures 4.8a and 4.9a show the normalised QoS of VLC streaming server when co-located with CPUBomb and Twitter-Analysis respectively. The minimum transcoding rate for the

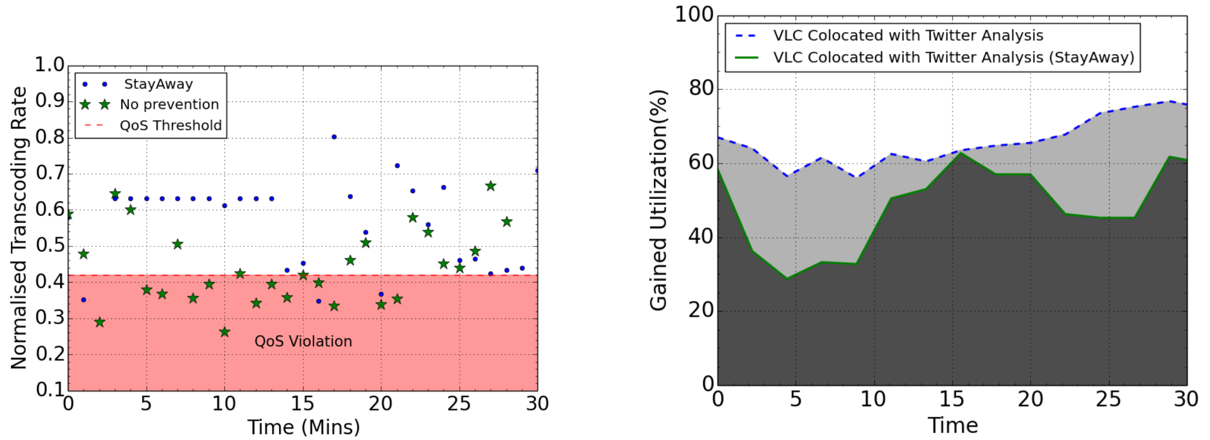


(a) VLC with CPUBomb (b) Gained Utilisation with CPUBomb

Figure 4.8: QoS violations of VLC streaming and utilization gained by co-locating VLC streaming alongside CPUBomb

VLC server to ensure uninterrupted delivery of frames is shown as QoS threshold. Whenever the rate of transcoding falls below this threshold the clients experience degradation in the quality of service. We can see from figure 4.8a and 4.9a that without any prevention the system experiences numerous violations as both these batch applications contend for resources with VLC streaming. The qualitative effect of Stay-Away on QoS becomes clear when reproducing the streamed video. Qualitatively, a choppy reproduction without Stay-Away transforms into a smooth playback when including Stay-Away. The QoS degradations are considerably reduced and most violations seen are in the early phase of execution. This is because the system is unaware of the states that correspond to a violation in the early phase and once seen, the system proactively prevents future violations. Other violations arise from factors such as instantaneous jumps to violation states characterised by sudden increase in the use of CPU.

Figures 4.8b and 4.9b show the gain in machine utilisation from co-location. Gained utilisation is the gain in utilisation in comparison to executing VLC streaming service without any co-location. The upper band in the figure shows the maximum utilisation that can be gained by co-locating the batch application with VLC streaming service without any prevention from performance interference. With Stay-Away deployed, we are able to achieve a good balance



(a) VLC with Twitter-Analysis

(b) Gained Utilisation with Twitter-Analysis

Figure 4.9: QoS violations of VLC streaming and utilization gained by co-locating VLC streaming alongside Twitter-Analysis

in improving machine utilisation, shown in the lower band from figures 4.8b and 4.9b, while still guaranteeing a high level of QoS. The gain in machine utilisation depends on the characteristics of the co-located batch application. In our setup, VLC streaming with Twitter-Analysis gains an average of 50% machine utilisation when compared to an isolated run of the VLC streaming server. The system gains substantial improvement in both utilisation and QoS. This is because Stay-Away throttles only when the system progresses toward resource contention. Co-location with CPUBomb as the batch application is the worst case scenario since the batch application constantly contends for CPU and does not experience any phase transition. As a result the gained machine utilisation, as shown in figure 4.8b, is in spikes as Stay-Away throttles in an attempt to mitigate performance degradation. The gain in utilisation for CPUBomb is about 5% because CPUBomb constantly consumes CPU and it is impossible to execute both VLC streaming and CPUBomb without violating the QoS. However, with Stay-Away deployed, it learns this contention and guarantees a high level of QoS.

Figures 4.11, 4.12 and 4.13 show the QoS achieved when different batch applications are co-located with Webservice for different types of workload. We can see that with Stay-Away, a high level of QoS is guaranteed. Figure 4.10

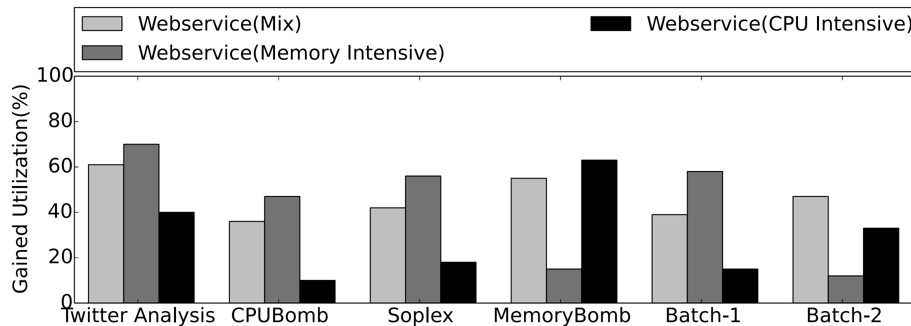


Figure 4.10: Gained Utilization when Webservice is co-located with different Batch Applications

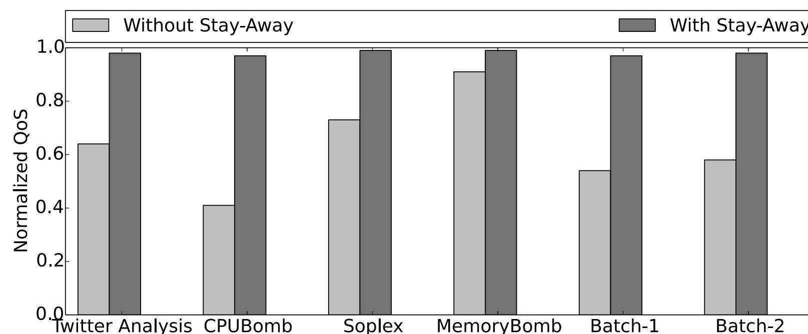


Figure 4.11: QoS of Webservice with a mix of CPU and Memory intensive workload when co-located with different Batch Applications

shows the gained utilisation when Webservice is co-located with different batch applications. The gained utilisation is different for different types of workload and the gain is maximum when Twitter-Analysis is co-located with Webservice for a memory intensive workload. This is because Twitter-Analysis experiences a mix of both CPU and memory intensive phases, and is throttled only during its memory intensive phase. The effect of performance interference caused by Twitter-Analysis is seen only when its memory operation is intensive enough to force the OS to swap pages of Webservice to disk, causing a degradation in response time. As a result, Twitter-Analysis is throttled only when it performs extensive memory operations. The gained utilisation is relatively low when batch application is co-located with CPU intensive workload of Webservice because all batch applications except MemoryBomb are mostly CPU intensive and interfere negatively with the performance of Webservice.

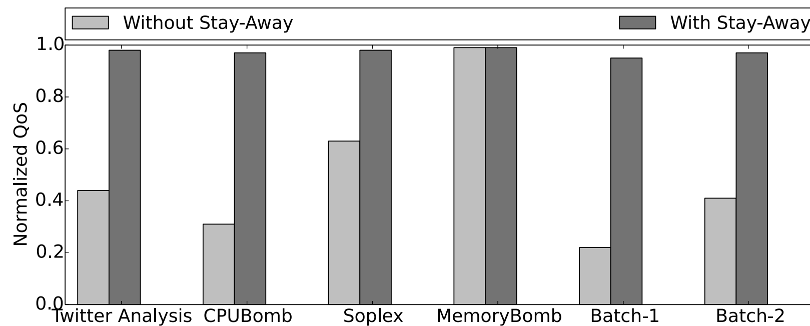


Figure 4.12: QoS of Webservice with CPU intensive workload when co-located with different Batch Applications

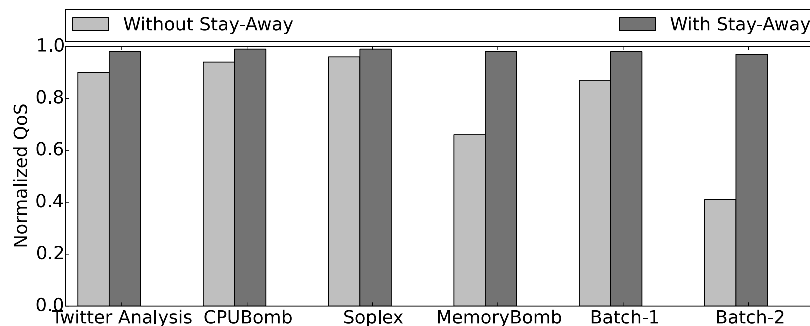
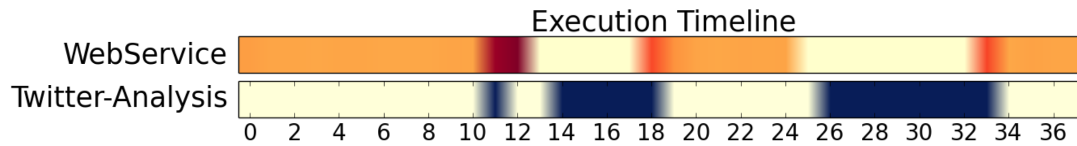
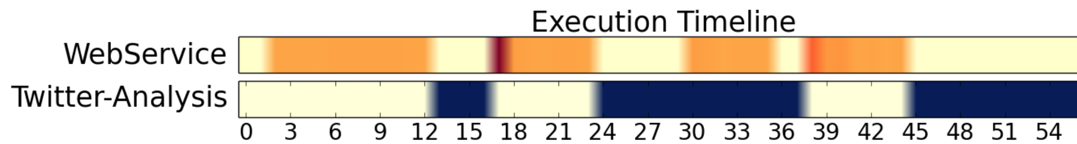


Figure 4.13: QoS of Webservice with Memory intensive workload when co-located with different Batch Applications

Figure 4.14a shows the execution timeline when Web service with a CPU Intensive workload is co-located with Twitter-Analysis. We vary the intensity of workload to show that Stay-Away is capable of detecting and using such periods of low utilisation without violating QoS. The stress on Webservice is measured by monitoring the number of transactions completed per second and is done only to illustrate the functioning of our middleware. Stay-Away does not have any access to this information. Twitter-Analysis begins execution at timestamp 10. This causes a stress on Webservice and leads to a QoS violation shown by a dark band. Stay-Away learns this and immediately throttles Twitter-Analysis. Soon after this, there is a period of low workload, which Stay-Away detects and the execution of Twitter-Analysis is resumed. Subsection 4.3.3 explains how Stay-Away detects this. At timestamp 18, the workload of Webservice increases and the execution of Twitter-Analysis begins



(a) Webservice(CPU Intensive) co-located with Twitter-Analysis



(b) Webservice(mix) co-located with Twitter-Analysis

Figure 4.14: The colour gradient for Webservice is a measure of the stress on its performance. Darker colour indicates higher stress. Dark colour bands for Twitter-Analysis represents period of its execution and lighter colour bands represents the period it is throttled.

to cause a stress on its performance, but hasn't violated QoS yet. Stay-Away predicts this and throttles Twitter-Analysis before a QoS violation happens. Figure 4.14b shows the execution timeline when Web service with a mix of CPU and memory intensive workload is co-located with Twitter-Analysis. We introduce a period of change in the workload phase from timestamp 30 to 36. We can see from the figure that Twitter-Analysis executes in an uninterrupted manner during this period as Stay-Away identifies the period as a phase change and believes that executing Twitter-Analysis would cause no stress. Stay-Away detects this from the state space mapping as the change in phase maps the corresponding states to a different space in the map and farther away from the violation state.

4.7.3 Template Validation

This section validates how a map generated for a latency sensitive application can be reused for future executions with a different set of batch applications. We conduct an experiment by streaming a video file using VLC running alongside CPUBomb as the batch application. The Stay-Away component is active during the run, capturing the states and preventing violation. Figure

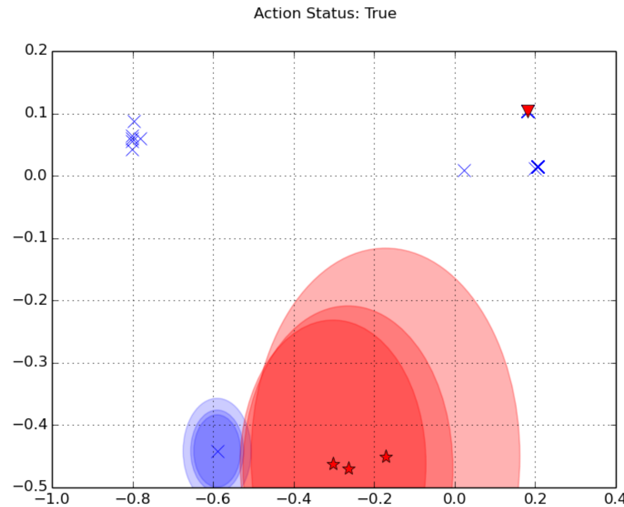


Figure 4.15: Template with CPUBomb

4.15 shows a snapshot of the states that characterises the VLC streaming service for a given video and is used as the template for future executions of VLC alongside a different batch application. In order to validate that the captured states correspond to the properties of VLC independent of the co-location with any specific batch application, we use the template as the initial state of VLC for streaming the same file alongside Soplex. We disable the Stay-Away component from taking any action to show that the states corresponding to violation in figure 4.15 continue to correspond to violation alongside Soplex. Figure 4.16 shows a snapshot of the state of VLC run alongside Twitter-Analysis. While new states are seen during the execution, we can see that there are more violations and they correspond to the area characterised by violations from figure4.15.

4.8 Summary

In this chapter, we design and demonstrate Stay-Away, a generic and adaptive mechanism to mitigate the detrimental effects of performance interference on sensitive applications when co-located with other batch-like applications and improve resource utilization. Unlike earlier previous work that requires apriori knowledge and static models, Stay-Away continuously learns and maps to a state-space representation the favourable and unfavourable states of execution

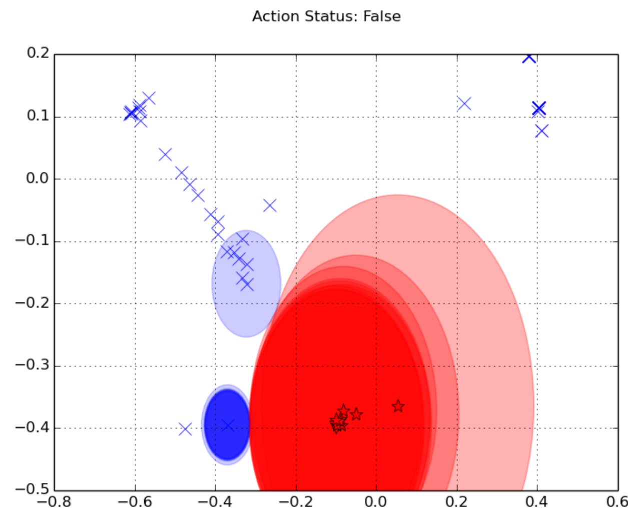


Figure 4.16: VLC with soplex

among multiple VM. The representation allows to visualise and interpret co-located VM execution. This is used to predict real-time transitions of the co-located VM states continuously and prevent performance degradation in selected VMs. Additionally, we discuss how this mechanism doubles as a template engine for repeatable experiments or services.

The evaluation of a proof-of-concept prototype of Stay-Away with Linux Containers and several batch and interactive applications confirm the expected effect, validity and stability of the mechanism in general.

Chapter **5**

Hubbub-Scale: Reliable Elastic Scaling Decisions

Abstract

Elastic resource provisioning is used to guarantee service level objective (SLO) with reduced cost in a Cloud platform. However, performance interference in the hosting platform introduces uncertainty in the performance guarantees of provisioned services. Existing elasticity controllers are either unaware of this interference or over-provision resources to meet the SLO. In this chapter, we specifically focus on the metrics used for elastic scaling decisions and show that assuming predictable performance of VMs to build an elasticity controller will fail if interference is not modelled. We identify and control the different sources of unpredictability and build Hubbub-Scale; an elasticity controller that is reliable in the presence of performance interference. Our evaluation with Redis and Memcached show that Hubbub-Scale efficiently conforms to the SLO requirements under scenarios where standard modelling approaches fail.

5.1 Introduction

Services that are elastically provisioned in the Cloud are able to use platform resources on demand. Instances can be spawned to meet the Service Level Objective (SLO) during periods of increasing workload and removed when workload drops. Enabling elastic provisioning saves the cost of hosting services in the Cloud, since Cloud users only pay for the resources that are used to serve their workload. Virtualization is a key enabler for elasticity as it ensures operational isolation for Cloud users and provides management convenience for Cloud providers. However, it does not provide performance isolation on many shared resources, such as memory sub-system. In other words, consolidation of multiple VMs comes at the price of application slow down and VM performance interference in ways that cannot be modelled easily.

The already hard problem of building a general-purpose elasticity controller that guarantees SLO with adequate resource provisioning becomes exacerbated by the role of performance interference. Previous works have proposed multiple models ranging from simple threshold based scaling [72, 73] to complex models based on reinforcement learning [74, 75], control modelling [76, 48, 45, 77], and time series analysis [44, 43] to drive the scaling decisions. While every model comes with its host of benefits and demerits, the impact of performance interference on elastic scaling is often overlooked.

CPU utilization [42, 43, 44, 45, 46] and workload intensity [47, 48, 49, 50, 51] are two widely used indirect metrics for elastic scaling since they are easily available and correlate well with the measure of service quality such as latency. In this chapter, we investigate if the performance interference from consolidation has a role to play on the metrics used for making scaling decisions. We find that it becomes imperative to quantify the contention in the system in order to achieve accurate scaling.

Our main contribution is Hubbub-Scale; an elasticity controller that achieves predictable performance in the face of resource contention without any significant overhead. We facilitate this by designing a middleware that provides an API to quantify the amount of pressure the co-running VMs put on the target system. Specifically our contributions are:

- We show that OS configuration, performance interference and power-saving optimisations stand in the way of predictable performance. While OS configuration and power-saving optimisations can be controlled, performance interference is inevitable in a multi-tenant system and needs to be modelled.
- In the presence of performance interference, indirect metrics used for elastic scaling cease to accurately reflect the measure of service quality, consequently affecting the scaling accuracy.
- In the presence of performance interference, even relying on direct metrics like latency to scale can lead to SLO violations.
- We build Hubbub-scale, an elasticity controller that is reliable in the presence of performance interference and achieves high resource utilization without violating the SLO.

5.2 Elastic Scaling

A typical elastic scaling process involves 2 steps: making a decision on when to scale and choosing the right number of instances to be added/removed to serve the changing workload. Figure 5.1 shows the components of an elasticity controller. The sensor gathers relevant performance metrics from the service which is then fed to the decision making module. The decision making module infers meaning from the gathered metrics to decide when to scale. The actuator is then responsible for carrying out the decisions of the decision making module and acts on the resource infrastructure. Both decision making and actuation depends on the capacity of a VM. *Capacity* is defined as the maximum amount of workload a VM can serve within a certain level of QoS. A decision to scale-out is established when the elasticity controller detects that the workload exceeds the current capacity of the VMs. It then determines the additional capacity needed to serve the excess workload and spawns the required number of VMs. The accuracy of scaling depends on the agility of the controller in detecting workload changes (decision making) and in satisfying the additional capacity (actuation). In this chapter, we focus only on the decision making phase, that determines when to make a scaling decision.

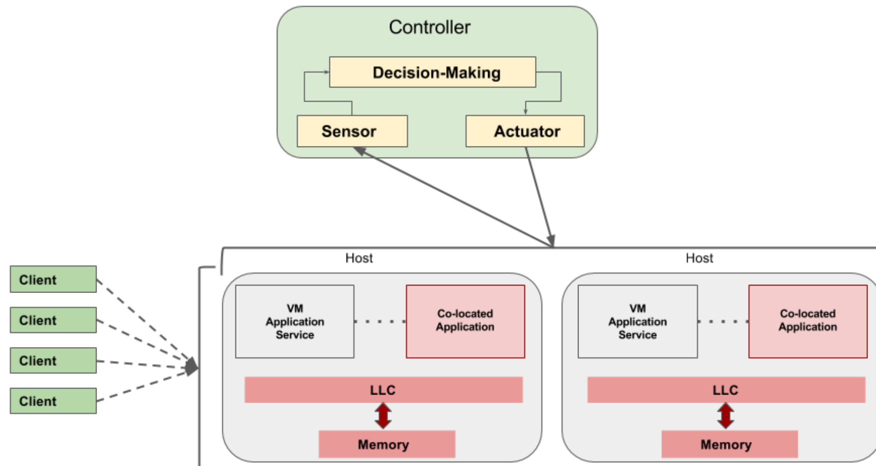


Figure 5.1: Architecture of a typical elastic scaling process

We give a brief background on load-based elastic scaling and the required properties for the metrics used to drive the scaling decision.

5.2.1 Scaling Type

Load-based scaling handle variable loads by starting additional instances when the workload increases and stopping instances when workload decreases, based on any of load metrics, such as request intensity (RPS). It can be achieved in three ways: reactive control, proactive control and a combination of reactive and proactive control. With reactive control, the system relies on reacting to changes in a system metric such as request intensity, intensity of I/O operations, CPU utilization, or direct metrics like latency to make scaling decisions. While this approach can scale the system with good accuracy, the system reacts to a workload metric change only after the change occurs and is observed. This may result in SLO violation if the reaction is too late. Proactive control, on the other hand, explores the historic access patterns of the workload, in order to conduct workload prediction and perform model-predictive control. With this approach, it is possible to prepare the instances in advance and avoid any disruption in the service when auto-scaling. Despite their respective merits and demerits, both approaches require run-time measurement of a metric to make the scaling decision and to drive the elasticity control.

5.2.2 Choice of Metrics

The right choice of metrics (control input) is critical for efficient elastic scaling since the performance, effectiveness and precision of the elasticity controller depends on the quality of the control input metric and the overhead in measuring and monitoring the control input [78]. In literature, authors have used a variety of metrics to make scaling decisions and to drive the elasticity control. An extensive list of those metrics is provided in [79]. A good choice of metric for the target environment should satisfy the following properties: (i) the metric should be easy to measure accurately without intrusive instrumentation because the controller is typically external to the guest application, (ii) the metric should be reasonably stable with little variations, (iii) should allow for quick reaction and (iv) the metric should correlate to the measure of level of quality of service (e.g, the service's average response time or latency) as specified in the SLO.

Direct Metric: A straightforward approach to scale is to directly rely on the metric (latency, response-time) specified in the SLO to drive the scaling decisions. However, it does not satisfy the properties of a good metric, since monitoring latency/response-time involves an overhead in measuring, needs instrumentation and reacts slower than an indirect metric. Some developers are however willing to incur the overhead in view of the benefits they accrue from easier scaling since response variable to be tuned is measured directly.

Indirect Metric: Scaling using indirect metrics do not measure the response variable directly, instead use other metrics that correlate well the measure of service quality (latency) and satisfies the properties of a good metric. CPU utilization is one such widely used metric [42, 43, 44, 45, 46]. It can be obtained from the operating system or the virtual machine without instrumenting application code. CPU utilization is also a more stable signal than metrics like response time and correlates well with the measure of service quality such as latency/throughput [45]. Another widely used metric for elastic scaling is workload in terms of Requests-per-second (RPS) [47, 48, 49, 50, 51]. RPS can be an important way to measure system performance and is mostly used for proactive control. Netflix developed a system called scryer [47] that uses workload to drive their proactive control for scaling decisions. Because CPU utilization and workload intensity are widely used in a large number of elasticity controllers, our work focuses on these two indirect metrics.

5.3 Motivation

There are 2 key aspects in elastic scaling that determine the effectiveness of the scaling model: when to add or remove instances, and, how many instances to add/remove. Any lapse in the accuracy of these two steps translates to SLO violations or increased cost from over-provisioning of resources. For example, a delay in adapting to an increasing load will result in SLO violations, and erroneously adding more instances than required will result in under utilized instances. All scaling models aim to minimise SLO violations and improve the resource utilization. However, existing scaling models fail to guarantee these properties in a multi-tenant setting. We explain this from two perspectives:

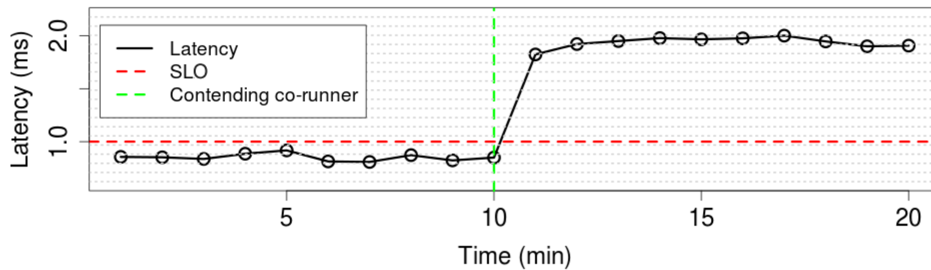


Figure 5.2: Variation of latency over time for a constant workload (RPS). Until 10 mins, the application runs in isolation. After 10 mins, other applications are executed on the physical host, generating contention at the shared system resources.

Using an indirect metric to scale: For ease of explanation, consider a simple control model that reacts to indirect system metrics to scale an application. The model learns the relationship between the system metrics (workload intensity (RPS), CPU utilization) and latency during a characterization phase, and decides when to add/remove instances and how many instances to add or remove in order to conform to its SLO. Figure 5.2 shows the latency of the application for a constant workload (RPS). After 10 mins, we introduce interference on the host and see that the model learnt by the elasticity controller becomes void and violates the SLO. This is because the system metrics do not accurately reflect the degradation caused by performance interference and the model is unable to discern the need to scale out. In other words, the system metrics correlate differently with latency in the presence of interference. It is

important to note that a dynamic learning model will also fail in the presence of interference as these metrics cannot attribute accurately for resource contention. Our experimental analysis in section 5.4 sheds more light on why this happens. Without quantifying the amount of contention in the system, it is not possible to achieve accurate scaling in a multi-tenant environment.

Using a direct metric to scale: In the presence of interference, even though the effect of degradation from contention is reflected directly on latency, and accurately aids the decision making phase, it can lead to over/under subscription of resources during the scaling phase. Without any additional information about contention, it is not possible to know the exact number of instances needed to scale the system.

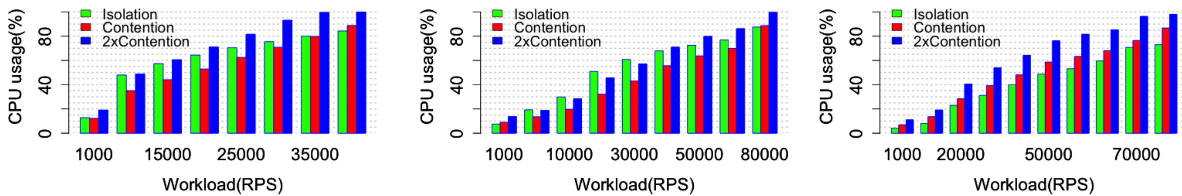
For example, first consider an isolated setting where an elastic application receives the workload W_c, handled by X instances, that corresponds to latency L_{SLO}. Without losing generality, we assume a round-robin load balancer. For an increased workload $W_c + n$, the total number of instances needed to ensure that SLO is not violated can be calculated as $\frac{W_c+n}{W_c} \times X$. Next, consider the case when the application is provided in a multi-tenant virtualized environment. In this case, the application can experience performance interference that causes performance degradation. The same latency L_{SLO} is then reached by a smaller workload $W_{ic} = W_c - \delta$ because of performance degradation. If the latency-driven elasticity controller is unaware of interference, it will allocate $\frac{W_c+n}{W_c} \times X$ instances to handle the increased workload $W_c + n$. This means that each instance will receive a workload greater than W_{ic}, thereby under subscribing to resources and violating SLO. Without quantifying the interference (I) and knowing W_{ic}, the elasticity controller cannot make accurate scaling decisions even when using a direct metric. There are further problems despite using a direct metric in a multi-tenant environment and is explained in chapter 6.

5.4 Experimental Analysis

Given the wide use of CPU utilization and workload intensity for driving the scaling decision, we set out to explore the reliability of these metrics in a multi-tenant environment. Our objective in this section is to answer if these metrics are reliable in the face of performance interference and to identify the different sources responsible for introducing unpredictability in the metrics.

To enable effective and accurate model control for elastic scaling, the control input (CPU utilization, Workload intensity etc) should be reliably predictable.

We perform experiments with Memcached [80] under a controlled setting. We execute SPEC CPU benchmarks [68] to create interference on the memory subsystem. The experiment is carried out in 3 phases: no interference (isolation), 1x contention (when 1 SPEC CPU benchmark instance runs alongside Memcached) and 2x contention (when 2 SPEC CPU benchmark instances run alongside Memcached). We study Memcached in progressive configurations, beginning with default settings of the kernel and later moving toward customized configurations, identifying the different sources of unpredictability and progressively ameliorating them. We eventually reach a configuration, where interference is the only source of unpredictability for CPU utilization and workload based model. In the rest of this chapter, we use the terms contention and interference interchangeably.



(a) Interrupts served on its own core (Config 1)

(b) Interrupts served by a different core (Config 2)

(c) Power saving optimisations disabled (Config 3)

Figure 5.3: CPU utilization (%) vs. Workload intensity (RPS) for Memcached with and without co-location for different configurations. Config 3 brings down the unpredictability solely to the intensity in contention and the number of co-located VMs.

We begin with the default kernel configuration and measure the variation of CPU utilization for different workloads when all the network interrupts of Memcached are served by the cores running the Memcached service. In the default settings (figure 5.3a), the interrupts were served on the core running the Memcached service. The cores responsible for handling Memcached spends most of its cycles processing network interrupts and this impacts the CPU utilization. To eliminate the additional impact of serving all the network

interrupts, the interrupts are configured to be served exclusively by a different core. Figure 5.3b shows the results for this configuration. Comparing figure 5.3b and figure 5.3a, it is evident that the maximum amount of workload that can be served before the CPU saturates is different under each setting. When interrupts are served by a different core, the application is able to handle a much higher workload before CPU saturation. It thus becomes imperative to have the same interrupt schedule enabled before a control model based on CPU utilization is designed for elastic scaling.

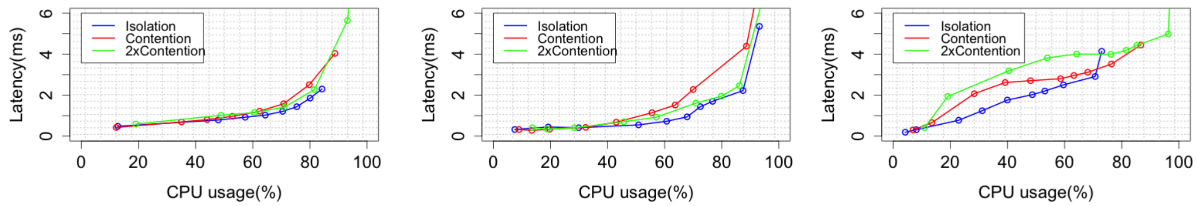
However, even with a core exclusively serving the interrupts (Figure 5.3b), the CPU utilization for a given workload, in some cases, drops in the face of resource contention. This is counter-intuitive because contention at the memory subsystem and the consequent delay results in increased processing time which should reflect in the form of increased CPU utilization. Somehow contention seems to improve the latency at reduced CPU usage. Upon investigation we found that at lower utilization levels, hardware power saving optimizations come into play and cause this behaviour. Our server, like nearly all machines today, incorporates several CPU power saving optimizations, such as idle power states and frequency scaling. These optimizations save precious energy resources and cause a counter-intuitive effect. We show in figure 5.3c that when power saving optimisations are disabled, this effect disappears and CPU usage increases with contention.

Once power saving optimisations are disabled (config 3), we see a more predictable correlation between CPU utilization and workload intensity (RPS). For a given workload, as the intensity of contention increases, the CPU utilization increases as expected. However in the following subsection, we explain why despite this predictability, elasticity controller still needs to be aware of contention in order to achieve accurate scaling.

5.4.1 CPU Utilization vs. Latency

In this section we show that although CPU utilization reflect the measure of service quality (latency) when running in isolation, they correlate differently in the presence and absence of interference.

Figure 5.4 shows the variation of CPU utilization with latency for different configurations. Each point in the graph corresponds to a specific workload.



(a) Interrupts served on its own core (Config 1)

(b) Interrupts served by a different core (Config 2)

(c) Power saving optimisations disabled (Config 3)

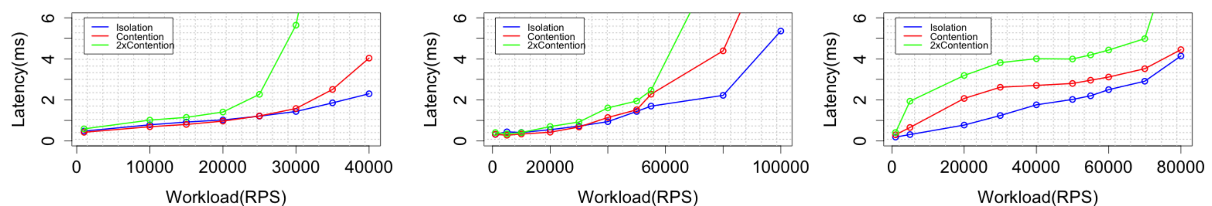
Figure 5.4: Variation of Latency (ms) vs. CPU utilization (%) in Memcached with and without co-location for different configurations. Config 3 brings down the unpredictability solely to the intensity in contention and the number of co-located VMs.

Although the latency remains fairly similar for smaller workloads (lower CPU utilization) under different levels of contention, they begin to deviate as CPU utilization increases. As CPU utilization increases, contention begins to have a significant effect on the CPU time spent for serving the requests and, as expected, latency increases. However, in config 1 and config 2, 2x contention performs better than 1x contention (labelled as contention in figure). Similar to our observations in the previous section, this randomness is ameliorated in config 3 by disabling power saving optimizations and higher amounts of contention corresponds to higher latency. However, despite this predictable behaviour, CPU utilization remains unreliable for making scaling decisions because the elasticity controller is unaware of contention in the system as it is external to the VM and cannot attribute the CPU cycles affected by interference. For example, in figure 5.4c, consider that the application has a service level objective of guaranteeing requests within a latency of 3ms. When building a model in an isolated environment, we learn that the system hits 3ms limit around 70-75% CPU utilization. Even by a conservative estimate, if the CPU threshold for scaling out is set to 65%, the system would experience SLO violations under 2x contention. we can see that in the presence of 2x contention the SLO (3 ms) would be violated at 40% CPU usage. The elasticity controller only takes as input the CPU utilization and is unaware of the presence of interference. Since the elasticity controller is unaware of the contention, even with an online learning model, it becomes challenging to make an accurate

decision. For the same latency value of 3ms, CPU utilization varies between 40% and 75% and ceases to reflect the measure of service quality.

Observation: *In the presence of performance interference, CPU utilization ceases to accurately reflect the measure of service quality if the amount of contention is not taken into account.*

5.4.2 Workload-Intensity (RPS) vs. Latency



(a) Interrupts served on its own core (Config 1)

(b) Interrupts served by a different core (Config 2)

(c) Power saving optimisations disabled (Config 3)

Figure 5.5: Variation of Latency (ms) versus Workload (RPS) in Memcached with and without co-location for different configurations. Config 3 brings down the unpredictability solely to the intensity in contention and the number of co-located VMs.

Figure 5.5 shows the variation of latency with increasing intensities in workload. In the previous section, we discussed why CPU utilization becomes unreliable in the face of performance interference. We can draw similar conclusions from this figure and clearly see that the amount of workload that can be handled within a certain latency bound varies significantly in presence of performance interference. The intensity in workload ceases to reflect the level of service quality in the presence of performance interference. In figure 5.5c, consider for example an SLO of guaranteeing requests within 2 ms latency. A proactive elastic controller can model the workload and predict in advance the decision of when to scale up/down an instance based on the incoming load. The proactive controller has to know when (at what workload) it should trigger a scaling action based on the prediction of the workload. However, with no information on the amount of contention experienced from the co-located VMs, it becomes almost impossible to make this decision accurately. This is because 5000

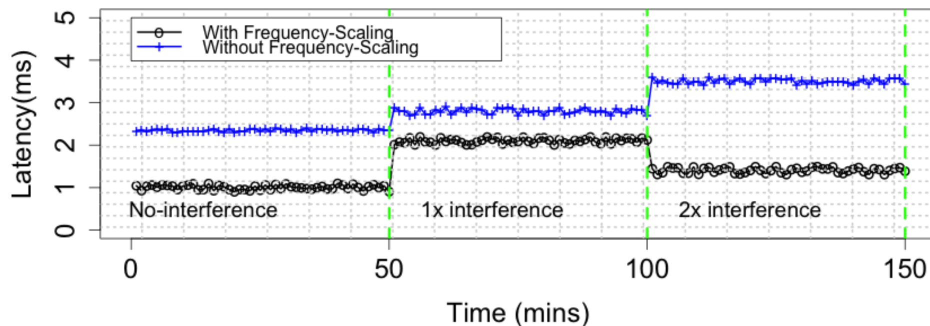
requests per second (RPS), 20000 RPS and 55000 RPS, all correspond to same SLO of 2ms. While a request of 20000 RPS doesn't violate the SLO when the service runs in isolation, the same workload violates the SLO in the presence of 2x contention. The controller is unaware of the interference experienced and cannot make an efficient decision.

Observation: *The maximum amount of workload that can be handled within some SLO bound varies significantly in presence of performance interference and is dependant on the behaviour of the co-located VMs.*

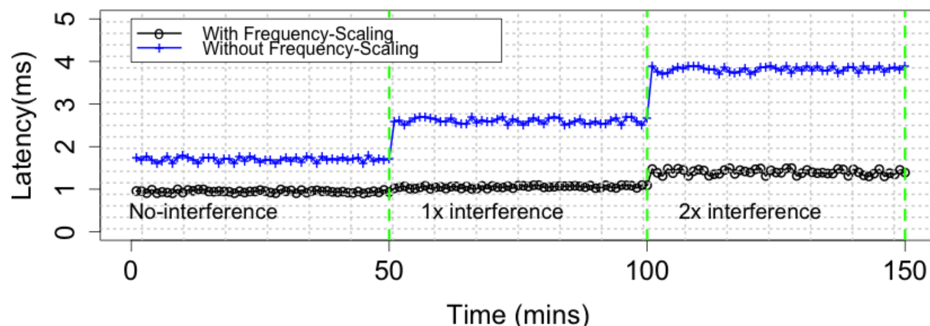
5.4.3 Putting them in perspective

In the previous subsection, we made a few observations from the experimental analysis. CPU utilization and workload intensity correlate well with latency in an isolated environment and can be reliably used for elastic scaling decisions. However, in a multi-tenant setting this correlation gets skewed. Our major goal was to understand how CPU utilization and workload intensity can be used reliably for elastic scaling decisions even in a multi-tenant setting. To this end, our first observation was that network interrupts and frequency scaling introduces randomness in the presence of contention. We also show in Figure 5.6a and figure 5.6b that when frequency-scaling is enabled, the CPU driver can scale the frequency of the processor depending on frequency governor enabled on the operating system resulting in unpredictable performance. With frequency scaling enabled in figure 5.6a, we see that under 2x contention Memcached performs better than 1x contention. Therefore to achieve our goal of using CPU utilization and workload intensity for scaling in a multi-tenant environment, the first step is to eliminate this randomness and we do this by disabling power saving optimisations. Once the frequency of the processor is fixed (blue line in figure 5.6a and figure 5.6b), Memcached behaves in a predictable manner with latency acting directly proportional to the amount of contention. Once this randomness is controlled, we can see that these metrics can be used for elastic scaling if the level of contention can be accounted for.

The observation that latency increases as contention increases indicates that the correlation between input metrics and latency gets skewed in the presence of interference. i.e., for the same amount of workload intensity/CPU utilization, Memcached can experience different latencies. Without quantifying contention it is not possible to attribute this variation in correlation. Therefore, to achieve



(a) Variation of latency when CPU utilisation is maintained between 65 and 70%



(b) Variation of latency when Workload (RPS) is fixed at 40000 RPS

Figure 5.6: Frequency-scaling affects performance predictability. Once frequency-scaling is controlled, performance variation from interference becomes predictable as shown by blue line.

our goal it becomes imperative to not only take into account interference but also be able to quantify the same in order to make reliable scaling decisions. Performance interference just like frequency scaling skews the correlation unless modelled. Table 5.1 provides a summary of the different sources of variation, their impact on modelling and potential ways to minimize this unpredictability. The metrics (CPU utilization, Workload intensity) achieve more predictability when all the sources of variation are controlled. However, minimizing the unpredictability comes with significant trade-offs and not all parameters can be completely controlled. Contention is one such source as it manifests in multi-tenant environments. While frequency-scaling and interrupt

processing can be controlled, performance interference is unavoidable in a multi-tenant setting and needs to be accounted for.

Source of Variation	**How to minimize variation**	**Trade-offs involved**	**Impact on Modelling**
Interrupt Schedule	Assign dedicated cores for interrupts	Lower throughput when running at low utilization	Minimal, since the interrupt schedule remains the same throughout the life cycle of the service
Frequency Scaling	Disable C-states and P-states	Results in increased power usage	High, Affects both CPU and Workload based modelling.
Contention	Over-provision resources	Results in under-utilized machines	High, Affects both CPU and Workload based modelling

Table 5.1: Source of variation in CPU Utilization and Workload based modelling

5.5 System Overview

We say that the storage system *co-runs* with other applications when they all run on different cores of the same physical host; we refer to all these applications as *co-runners*.

If we are able to model contention, then it is possible to attribute the variation in correlation and still rely on these input metrics to achieve reliable scaling. Contention can be seen as the amount of pressure an application puts on different shared resources such as memory-bandwidth or the cache. Each application may have a different amount of cache and memory-bandwidth usage and this determines the contentiousness of the application. However, sensitivity to contention depends on the application's reliance on the shared memory subsystem and how much an application progress benefits from this reliance. Contention and sensitivity need not always be correlated [34] and they need to be modelled separately during the run-time. In our case, we are interested in the contentiousness of the co-runners and the sensitivity of the

target system. Prior works [81, 82, 35, 83] use the target systems last level cache (LLC) miss rate/ratio as an indicator to detect contention and classify application for contention aware scheduling. While LLC miss rate/ratio can be a good indicator of contention, it suffers from the following limitation: An application can have varying run-time behaviour and depending on the application access patterns, LLC misses can vary over time making it difficult to attribute if contention is the sole cause for LLC misses. While it may be possible to detect contention in some cases based on LLC misses, it still cannot quantify the amount of degradation.

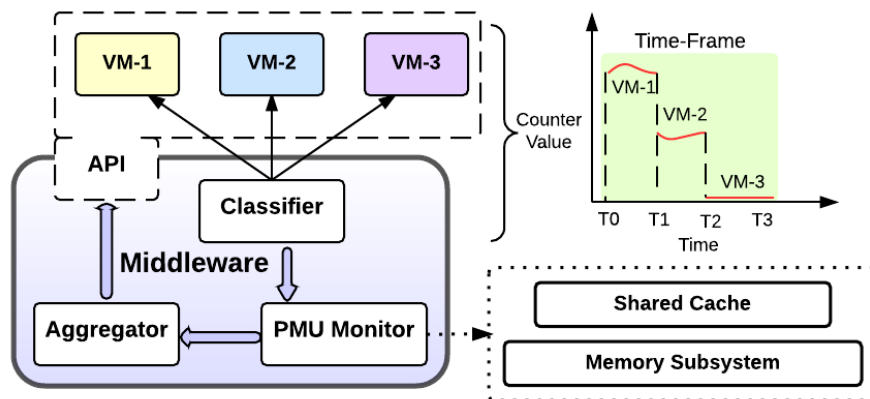


Figure 5.7: Architecture of the Middleware

PMU based approximation: For the reasons mentioned above, instead of relying on the behaviour of the target system alone, we take into account the co-runners behaviour for quantifying the contention. We use performance monitoring units (PMU) to approximate this behaviour. PMU's are special registers in modern CPUs that can collect low level hardware characteristics of an application without any additional overhead. The goals are two-fold: to identify the existence of contention from the co-runners and to quantify the amount of the pressure exerted on the target system. It is important to quantify the amount of pressure exerted by the co-runners since this has a direct impact on the amount of performance degradation suffered by the target system. Different amount of contention causes different amount of degradation.

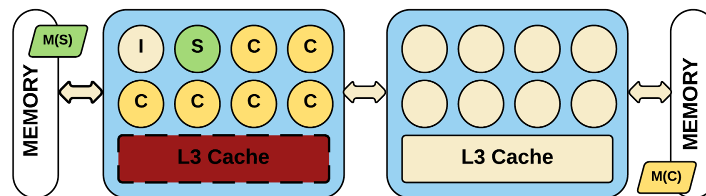
Middleware: Figure 5.7 shows the architecture of the middleware to quantify contention on the memory subsystem. The middleware provides an API that can be queried to access information about the contention from the

co-runners. The different components of the middleware provide the following function: The classifier is responsible for identifying the VMs that need to be monitored for contention. It optimises the number of co-runners to be monitored. The role of the classifier is to only select those VMs that are potential candidates for creating contention at the memory subsystem. This minimizes the overhead of unnecessarily analysing VMs that are idle or not memory-subsystem intensive. The classifier maintains a moving window of the average CPU utilization of different VMs and selects only those VMs that consistently have a CPU utilization over a certain threshold. This is because any application that is intensive on the memory-subsystem has a high CPU utilization. In our experiments we set our threshold to 30%. The list of selected VMs are then passed on to the PMU Monitor. The PMU Monitor monitors the performance counters of the VM using the "burst-approach" as explained in the next section. The measured counter values are then passed on to the aggregator that calculates a metric called the interference-index (explained in section 5.6) . Interference-index approximates the pressure the co-runners put on the target system. The aggregator subsequently makes them available for the API along with the monitored counter values to allow for a user specific composition for quantifying contention. By exposing different counter metrics through the API, it also allows the users to compose their own index of pressure for any subsystem.

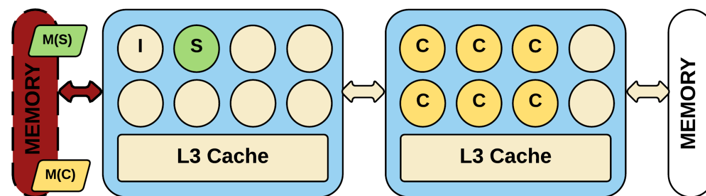
Burst-Approach: Typically performance counters are saved/restored when a context switch changes to a different process, which costs a couple of microseconds. Since these counters can hold context for only a single process at a time, monitoring the behaviour of the co-runners during runtime requires the middleware to adapt to this limitation. We circumvent this limitation by using a "burst-approach" where different co-runners are monitored in bursts and their values composed together within a single time-frame. A time-frame is defined as the period during which an application is assumed to have minimal variations in its behaviour. Consider, for example 2 VMs co-located on a physical host. In order to monitor their behaviour, the middleware collects the counters of each VM one after another in cycles. The time chosen to measure the counters of all the VMs exactly once defines a time-frame. Figure 6.4 shows one time-frame of execution. It is important to ensure that the chosen time-frame is neither too long nor too short. A very short time-frame can leave no time for the counters to be monitored since releasing and reacquiring

the counters costs a couple of microseconds. Also, very short durations do not accurately capture the application behaviour. On the other hand, long time-frames aren't ideal either because it increases the probability of variation in the application behaviour and violates the assumption that the application experiences minimal variations. This is important because the behaviour of every co-runner is composed together each time-frame. Major variations in the application behaviour during a time-frame can thus result in misleading conclusions.

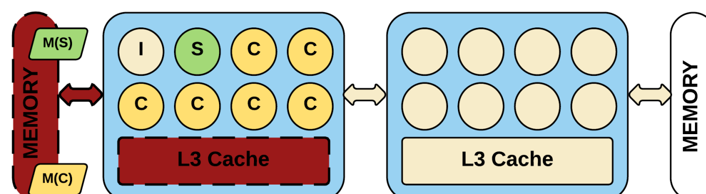
5.6 Characterising Contention



(a) C's data is remote, hence S contends with C's only for L3 cache (config-a)



(b) C runs on a different processor, hence S contends with C's only for memory bandwidth (config-b)



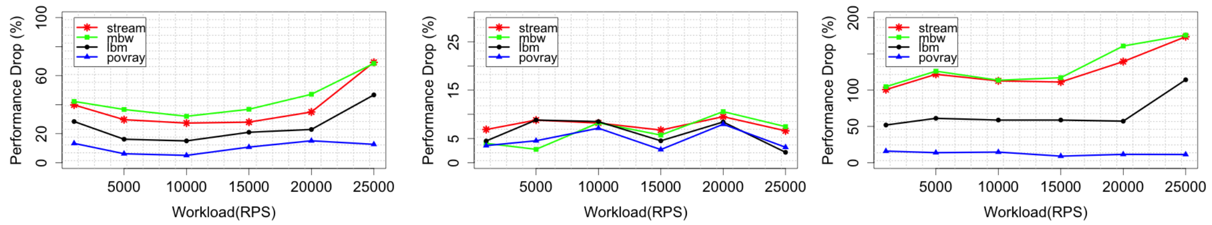
(c) S and C share both memory bandwidth and L3 cache and hence contend for both (config-c)

Figure 5.8: Configurations for generating contention at different resources. S denotes the storage system and M(S) denotes the memory allocation of S. C denotes the co-runners and M(C) denotes the memory allocation of C. I denotes the core serving interrupts.

In order to characterise contention, we choose in-memory storage systems, i.e. Memcached and Redis, as demonstrative target systems to show the scaling of services under performance interference. We identify which resources, upon contention, degrade the performance of the storage system and the properties of the co-runners that determine the level of contention. In order to understand the properties of the storage systems and the memory sub-system they are sensitive to, we characterise them on a NUMA machine.

We denote the storage system by S and each of its co-runners by C. In our experiments, we compute the performance drop as follows: First, we measure the average latency L_i of the storage system when running in isolation. Then we measure the average latency L_c of the storage system when it co-runs with other processes. Performance drop suffered is $\frac{L_c - L_i}{L_i}$.

5.6.1 Sources of degradation



(a) Contention for L3 cache (config-a)

(b) Contention for memory bandwidth (config-b)

(c) Contention for both resources (config-c)

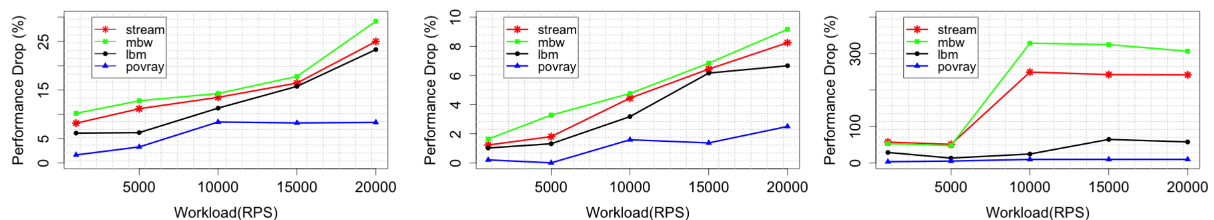
Figure 5.9: The drop in performance of Memcached for different throughputs. Memcached is run alongside 6 instances of different co-runners.

There are 2 main subsystems responsible for contention: the cache and the memory bandwidth. In order to assess the impact of contention on these subsystems, we use three system configurations illustrated in figure 5.8. They are designed to generate contention at different resources: the first configuration generates contention only on the cache, the second only on the memory bandwidth and the third one on both. Figure 5.9 and 5.10 show the drop in performance experienced by Memcached and Redis respectively.

For both Memcached and Redis, it is clear that cache is the dominant source of performance degradation. In the case of Redis, cache contributes to a maximum

of 30% drop in performance (figure 5.10a) while bandwidth only causes 8% (figure 5.10b). Similar observations can be made for Memcached, with cache contributing upto 65% drop in performance and bandwidth contributing less than 10%. However, the over all drop in performance drop of Redis is much higher in comparison to Memcached. Upon deeper analysis, we found that beyond 10000 RPS, Redis reaches a point of saturation in terms of available CPU. With proper configuration and optimization, it is possible to improve the throughput of Redis much beyond this limit. Since our intent is to demonstrate the impact of interference, we do not consider optimization or configuration set up to improve throughput. The results show that both the storage system benefits more from it's reliance on the cache than from memory bandwidth.

Our results are related to the conclusion drawn by running packet processing workloads on multicore platforms. The dominant contention source was found to be the cache [17]. As we will show, the difference comes from our observation that memory access pattern also impacts the performance of in-memory storage systems. On the contrary, SPEC CPU benchmarks are more sensitive on memory bandwidth [34].



(a) Contention for L3 cache (config-a)

(b) Contention for memory bandwidth (config-b)

(c) Contention for both resources (config-c)

Figure 5.10: The drop in performance of Redis for different throughputs. Redis is run alongside 6 instances of different co-runners.

5.6.2 Properties that determine degradation

We investigate properties of the co-running application that cause performance degradation. In both figures 5.9 and 5.10, all the different co-runners cause degradation in the same order; ie. mbw consistently causes the highest amount of performance degradation, followed by stream, lbm, and povray. In order to understand the properties that define the aggressiveness of the co-

Co-runner (6X)	Cache References (millions)	Cache Misses (millions)	L3 Prefetch (millions)	L3 Prefetch miss (millions)	Memory bandwidth (GB/s)
mbw	150.1	54.1	73.76	68.29	20.3
stream	133.2	47.3	107.2	98.6	20.5
lbm	63.1	25.5	135.1	102.2	18.8
linearwalk	228	84.6	150.5	91.2	22.4
libquantum	242.5	91.6	93.3	57.7	21.2
randomwalk	1055.5	137.7	0.165	0.132	8.4
povray	24.6	0.015	37.6	0.009	0.01

Table 5.2: Memory-subsystem behaviour of co-runners sorted by performance drop (highest to lowest) experienced by Redis and Memcached

runners, we rely on PMUs. L3 cache-references of the co-running applications were consistent with our observation and appears to mostly determine the degradation suffered. mbw has the highest number of cache-references and povray the lowest. This makes sense because higher cache references from the co-runners effectively reduces the cache space of the storage system, resulting in a drop in performance.

However, table 5.2 (sorted by descending order of performance degradation) shows that cache-references alone does not determine the performance drop of the storage system. For example: randomwalk, linearwalk and libquantum have higher cache references than mbw, but they cause much lesser degradation. Linearwalk does a walk through the memory in a linear fashion being completely predictable, while randomwalk pseudo randomly walks within a page. Our results show that the sensitivity of the storage system also depends on the memory access patterns of the co-runner. We designed the application linearwalk and randomwalk precisely to study this property. Cache references do not capture the memory access patterns of the application. However, cache misses along with prefetch misses can provide hints about the memory access patterns of the application. In order to identify all relevant counters that affect

the sensitivity of the storage system, we run a typical feature selection process that evaluates the effect of different performance counters on the sensitivity of the storage system. The chosen performance counters are shown in 5.3. The chosen metrics correlate well with our observation on memory access patterns and cache access intensity of the co-runner. The feature selection process however indicated that DTLB loads and stores of the co-runner also reflect the sensitivity of the storage system. When we quantify the interference index we found that the impact of DTLB loads and stores are minimal and cache-references, cache-misses, prefetch and prefetch-misses are the strongest indicators to model sensitivity. Our model also includes cache-references of the target system to take into account different workloads. It inherently allows to generate a performance degradation model for different workloads. In the following section we explain the process of constructing the measure of degradation (interference-index) from these performance counters.

Summary: *Although the dominant contention factor is the cache, sensitivity of the storage system is not determined only by the number of cache references of the co-runners. The memory access pattern of the co-runner plays a significant role in determining the performance of the storage system.*

5.6.3 Interference-Index

The goal of characterising contention is to quantify the properties of the co-runners that lead to performance degradation of the storage system. We call this metric interference-index and it approximates the performance degradation suffered by the storage system. In order to be useful for elastic scaling, the metric must correlate with the performance drop suffered by the storage system.

We derive a set of N representative performance counters $WS = m_1, m_2, ..., m_N$ where m_i represents the metric i. For applicability across heterogeneous machines, we rely only on generic counters to approximate this pressure. We find that even relying on a very coarse approximation to quantify contention can improve the accuracy of the scaling decisions significantly. Using the counters in 5.3 and a training data set of co-runners, our system then builds a model that correlates co-runner properties with performance drop suffered by the storage system. We then use linear regression on these counter to construct the interference index. Figure 5.11 shows the interference-index constructed

Name	Description	Name	Description
cpu-clk	Reference cycles	inst-retired	Instructions retired
cache-ref (Co-runner& Target-system)	References to L3 cache	cache-miss	L3 cache misses
llc-prefetch	L3 prefetches	llc-prefetch-miss	L3 prefetch misses
dtlb-loads	DTLB loads	dtlb-load-miss	load misses in DTLB that cause page walks
dtlb-store	DTLB stores	dtlb-store-miss	store misses in DTLB that cause page walks

Table 5.3: Performance counters included in characterising contention

for Memcached. Since the modeling is data-driven, the interference index generated is application-dependent. We however do not view this as an issue since modeling can be fully automated.

From figure 5.11, we see that interference-index correlates with performance drop suffered by the storage system. Higher the interference-index, greater the performance drop experienced by the storage system. Also similar interference-index should correspond to similar drop in performance. For example, mbw4 and stream4 indicate 4 instances of the co-runners mbw and stream respectively and are not included in the training set. stream4 causes similar degradation as linear walk and they both correspond to the similar interference indexes. mbw4 causes a degradation that is greater than linear walk but lesser than lbm and is also captured by the model as expected. We also test our model on a different set of co-runner, omnetpp from the SPEC benchmark. Our model is able to predict the drop with a good accuracy. Once interference-index is quantified, it is then used as a control input along with CPU utilization/workload intensity for the elasticity controller.

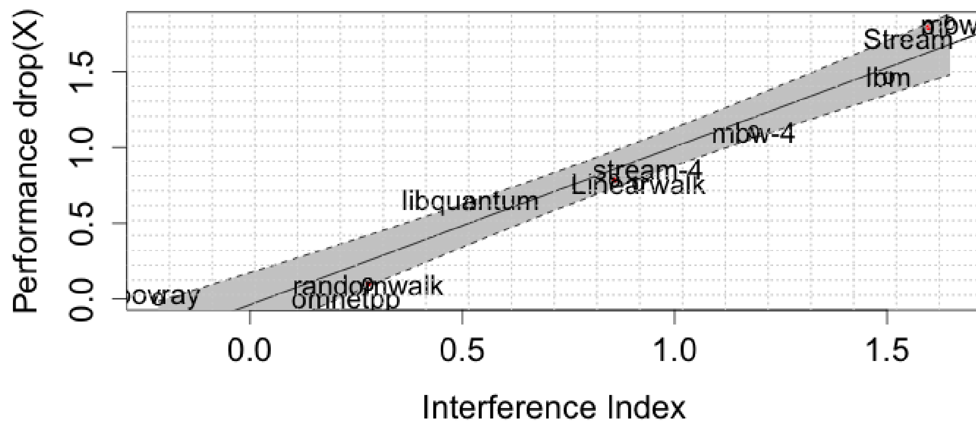


Figure 5.11: Interference-index quantifies the performance drop suffered by Memcached based on the behaviour of the co-runner.

5.7 Elasticity Controller

The main goal of an elasticity controller is to allocate adequate resource to a provisioned system in order to make the system operating in a healthy region that matches the control goal, e.g. Service Level Objective (SLO). The elasticity controller also optimizes the provisioning cost and prevents over-provisioning by allocating resources only when needed and freeing them when they are no longer needed. In our scenario, we define average service latency in small epochs (10 second) as our control goal, which is one of the common metrics specified in SLO between service providers and consumers. In the scenario of cloud computing, the amount of resources is translated to the number of virtual machines (VMs). As mentioned before, we target elasticity controllers that monitor system metrics, i.e. CPU utilization or incoming workload, i.e. read and write request rates, and model them as inputs. The former metric is commonly used to build an elasticity controller based on control theory [45, 72, 73] while the latter one is widely used to construct an elasticity controller using model-based control [51, 50, 48], such as Statistical Machine Learning (SML). Our observations from section 5.4 show that the number of VMs in the system cannot be directly and linearly translated to the capability of the system to handle workload. Specifically, handling a workload under SLO constraint requires different numbers of virtual machines depending on the presence and intensity of VM interference. In Hubbub-

Scale, we quantify the interference experienced in the system by querying the middleware API for interference index. Apart from this, the controller also takes CPU utilization and incoming workload intensity to model the load in the system.

Hubbub-scale is implemented as a centralized elasticity controller and its scaling decision is made by consulting control models that are built in an online fashion. There are two separate processes running in the Hubbub-scale controller that we call the model training process and the scaling process. Both processes run in parallel at different frequencies and do not interfere with each other. The model training process is used to continuously learn the application behaviours under different loads and intensities of interferences and update the control model. The load of the system is learnt from the sampled workload intensity or the CPU utilization on each VM. The monitored system behavior is narrowed down to the interested control goal, which is average service latency in small epochs. It is achieved by instrumenting the system to sample read and write latencies. We keep the monitoring overhead minimal by reducing the percentage of the sampled requests and the reporting frequency of the statistics to the model training process. The model training process updates the model with a simple data fading algorithm that uses weighted averages of service latencies from the most recent 10 epochs.

The scaling process consults the updated model with the monitored load of the system and the interference index from our API to make scaling decisions. To be specific, Hubbub-scale models the load of a system in two ways: workload-based modeling and CPU-based modeling The parameters used in workload-based and CPU-based models are firstly trained offline, and then improved during our online training process.

Workload-based Modelling: In workload-based modeling, the model is built and trained in a form of a binary classifier, which is widely used in recent works [51, 50, 48]. The classifier classifies the operational status of the modeled system. In our case, it models whether the system is operating in a state where its service latency violates the SLOs or not. Figure 5.12 shows a simplified version of Hubbub classifier, which uses finer data granularity, for explanation purposes. The binary classifier assumes that a certain VM is able to handle a specific load of the system, incoming read and write request rates, within the SLO constraint. The classifier is trained by having VMs operating

with different intensity of workloads and monitoring the achievement of SLO. For example, when training the model without interference, an operating state with SLO violated is marked as a red cross in Figure 5.12 and an operating state complying SLO is marked as a green dot. The boundary of the red crosses and green dots is learnt using SVM (Support Vector Machine), which forms the classifier (the black border). Hubbub-scale provisions the underlying system to be operated just under the boundary of the classifier to save the provisioning cost while satisfying the SLO. Hubbub-scale trains the classifier not only based on the incoming workload in terms of read and write request rate, but also takes into account the interference experienced on VMs, which is indicated from our interference index. Specifically, the amount of workload that can be handled by a VM is also learnt under different interference indexes with respect to the latency SLO constraint. The blue and pink borders in Figure 5.12 illustrate the learnt classifiers under 0.3 and 0.6 interference index. Thus, the binary classifier used in Hubbub-scale has 3 dimensions. It has an additional interference index dimension compared to the classifier proposed in [50], which has only 2 dimensions.

By obtaining the current workload and interference index, Hubbub-scale controller is able to calculate the number of VMs needed in the system using the following formula, where $AverageThroughputperServer$ denotes the throughput that can be handled by a server within the SLO constraint in the current level of the interference index.

$$NewNumberofServers = \frac{CurrentWorkload}{AverageThroughputperServer}$$

CPU-based Modelling: Hubbub-scale can also model system's load using CPU utilization on each VM. A classical integral controller is built because of its self-correcting and provably stable performance in the application of a wide range of scenarios, and has been used successfully in state of the art systems [45, 84, 85, 86]. The core of the integral controller is the following formula:

$$a_{k+1} = a_k + K_i * (y_{ref} - y_k) \tag{5.1}$$

a_k and a_{k+1} are continuous integers that represent system capability at current control period and the next control period, which is then translated to the number of VMs that are needed in the system. K_i is the integral gain parameter [85]. y_k is the current input and y_{ref} is the desired input. The

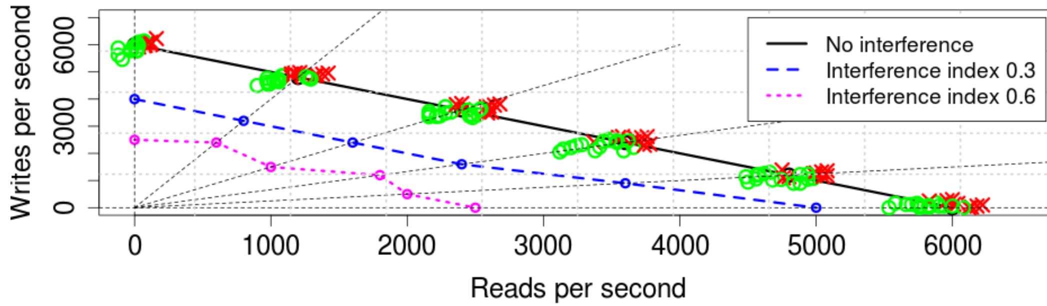


Figure 5.12: Throughput Performance Model for different levels of Interference. Red and green points mark the detailed profiling region of SLO violation and safe operation respectively in the case of no interference.

inputs are the monitored aggregated CPU utilization. Different values of desired CPU utilization y_{ref} are obtained with respect to a certain level of interference and the latency SLO. The controller obtains the desired VM numbers a_{k+1} from the previous time step a_k proportionally to the deviation between the current y_k and desired y_{ref} values of the CPU utilization in the current control period.

The difference between Hubbub-scale and a standard scaling approach is that Hubbub-scale takes into account the interference index in its model building. Standard scaling approaches rely on the standard modelling techniques, i.e., workload-based modeling and CPU-based modeling, used in the state of the art systems [50, 45, 51]. Specifically, standard workload-based modeling assumes a VM performs in an ideal scenario and is always capable of handling a specific workload without any knowledge of interference, similar to the implementation in [50, 48]. Standard CPU-based modeling only has one reference value (y_{ref}) in the model with respect to the latency SLO. In our evaluation, we show the inaccuracy of the standard modeling in the presence of interference and the accuracy of Hubbub-scale in conforming to SLO requirements.

Overhead: Our middleware has a very minimal overhead since it only samples the counters every few seconds. It has a negligible CPU consumption of less than 3% and does not perform any instrumentation to the application that result in performance loss. Hubbub-Scale incurs very negligible overhead in comparison to standard modelling approaches since the only additional

step involved is the construction of interference index, which in itself relies on non-intrusive monitoring.

5.8 Experimental Evaluation

We implemented our middleware on top of a KVM virtualization platform and conducted extensive evaluation using Memcached and Redis for varying types of workload and varying degrees of interference. This section describes our experiment setups and results.

5.8.1 Experiment Setup

All our experiments were conducted on the KTH private Cloud which is managed by Openstack [87]. Each host is an Intel Xeon 3.00 GHz CPU with 24 cores, 42GB memory and runs Ubuntu 12.04 on 3.2.0-63-generic kernel. It has a 12 MB L3 cache and uses KVM virtualization. The guest runs Ubuntu 12.04 with varying resource provisioning depending on the experiment. We co-locate memory intensive VMs with the storage system on the same socket for varying degrees of interference by adding and removing the number of instances. MBW, Stream and SPEC CPU benchmarks are run in different combinations to generate interference. In all our experiments we disable DVFS from the host OS using the Linux CPU-freq subsystem.

Our middleware performs fine-grained monitoring by frequently sampling the CPU utilization and the different performance counters for all the VMs on the host and repeatedly updates the interference index every 1 min. The time-frame chosen for monitoring the selected VMs after classification is 15 seconds and the counters are released for use by other processes for 45 seconds. The hosts running our experiments also run VMs from other users which introduces some amount of noise to our evaluation. However, our middleware also takes into account those VMs to quantify the amount of pressure exerted by them on the memory subsystem.

To focus on Hubbub-Scale rather than on the idiosyncrasies of our private Cloud environment, our experiments assume that the VM instances to be added are pre-created and stopped. These pre-created VMs are ready for immediate use and state management across the service is the responsibility of the running service, not Hubbub-Scale. Alternatively, interference generated

from data migration can be accounted for by the middleware to redefine the SLO border to avoid excessive SLO violations from state transfer. In order to demonstrate the exact impact of varying interference on Hubbub-Scale, we generate equal amounts of interference on all physical hosts and decisions for scaling out are based on the model from any one of the hosts. The load is balanced in a round robin fashion to ensure all the instances receive an equal share of the workload. We note that none of this is a limitation of Hubbub-Scale and is performed only to accurately demonstrate the effectiveness of the system in adapting to varying levels of workload and interference with respect to the latency SLO.

The control model of Hubbub-scale is partially trained offline before putting it online. Offline training is highly recommended but not mandatory. It identifies the operational region of the controlled system on a particular VM in an interference-free environment. Also, it improves the accuracy of the scaling during warm up phase. However, the Hubbub-scale control model can never be fully trained offline, because inter-VM interferences are hard to artificially produce as a cloud tenant. So, this part of the model can only get trained in an online fashion. The control models used in our evaluations are well warmed up by training them with different workloads and interferences online.

5.8.2 Results

Our experiments are designed to demonstrate the ability of Hubbub-Scale to dynamically adapt the number of instances to varying workload intensity and varying levels of interference, without compromising the latency SLO. The experiments are carried out in four phases, shown in figure 5.13 with each phase (separated by a vertical line) corresponding to a different combination of workload and interference setting. We begin with a workload that increases and then drops with no interference in the system. The second phase corresponds to a constant workload with an increasing amount of interference and later drops. The third phase consists of a varying workload with a constant amount of interference and in the final phase, both workload and interference vary. We carry out this experiment for 2 different types of control models: workload-based modelling and CPU-based modelling.

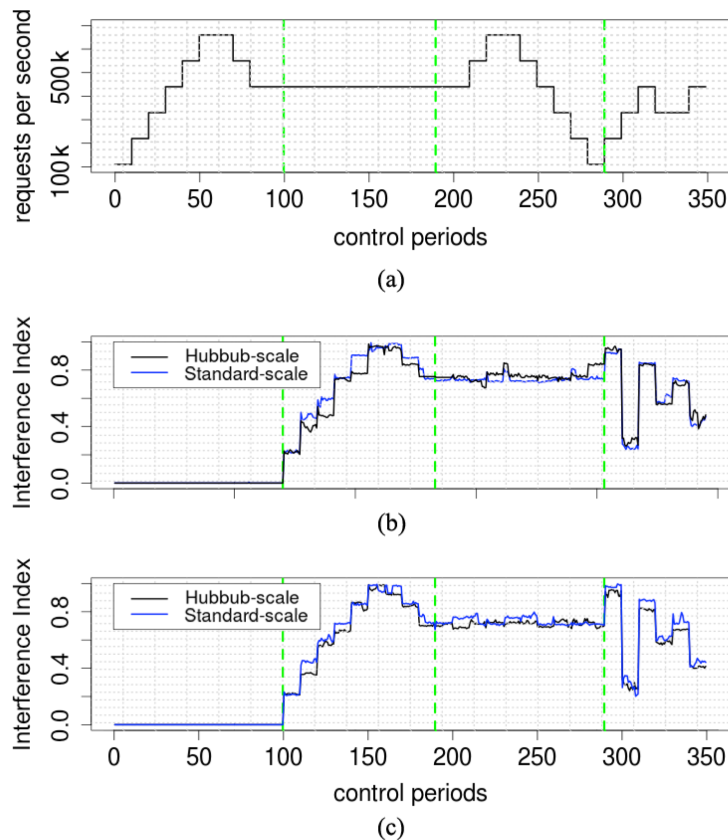


Figure 5.13: Experimental setup. The workload and interference are divided into 4 phases of different combinations demarcated by vertical lines. (b) is the interference index generated when running Memcached and (c) is the interference index generated when running Redis.

5.8.2.1 Scaling Out using a Workload based Model with/without Interference

Figure 5.14(b) and 5.15(b) compares the latency of a standard control model based on throughput performance modelling against Hubbub-Scale for all the four different phases for Memcached and Redis respectively. Without any interference (first phase), both systems perform equally well. However, in the presence of interference, the SLO guarantees of a standard control model begins to deteriorate significantly (figure 5.14(b), plotted in log scale to show the scale of deterioration). Hubbub-scale performs well in the face of interference and upholds the SLO commitment. The occasional spikes

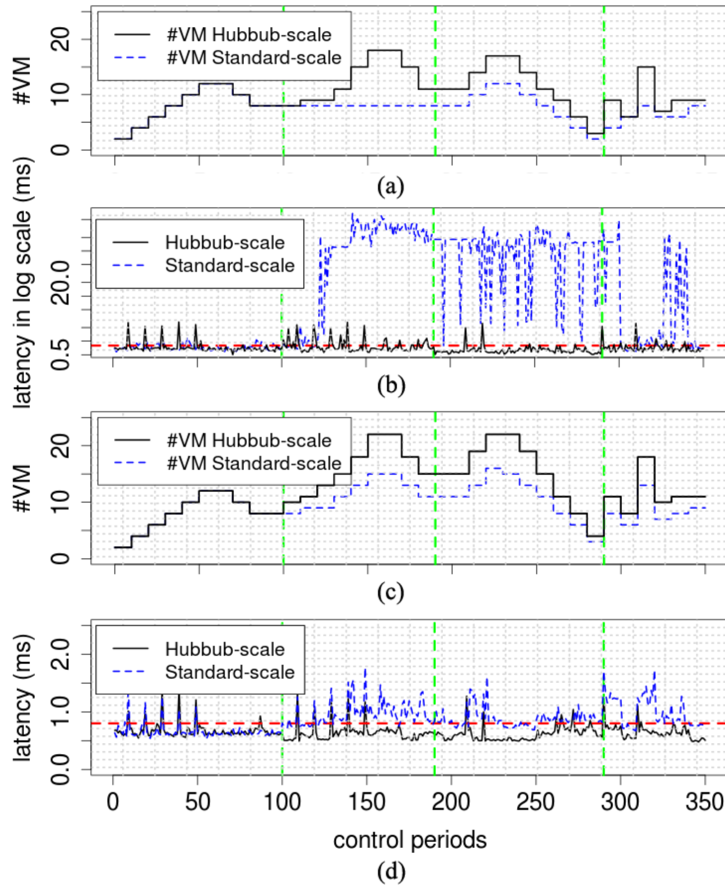


Figure 5.14: Results of running Memcached across the different phases. (a) and (b) shows the number of VMs and latency of Memcached for a workload based model. (c) and (d) hows the number of VMs and latency of Memcached for a CPU based model.

are observed because the system reacts to the changes only after they are seen. Figure5.13(b) plots the interference index captured by the middleware during the run-time corresponding to the intensity of interference generated in the system. The index captures the pressure on the storage system for different intensities of interference. Certain phases of the interference index in the second phase do not overlap because of the interference from other users sharing the physical host (apart from generated interference). We found that during these periods services such as Zookeeper and Storm client were running alongside our experiments increasing the effective interference generated in the system. Figure 5.14(a) and 5.15(a) plots the number of active VM instances

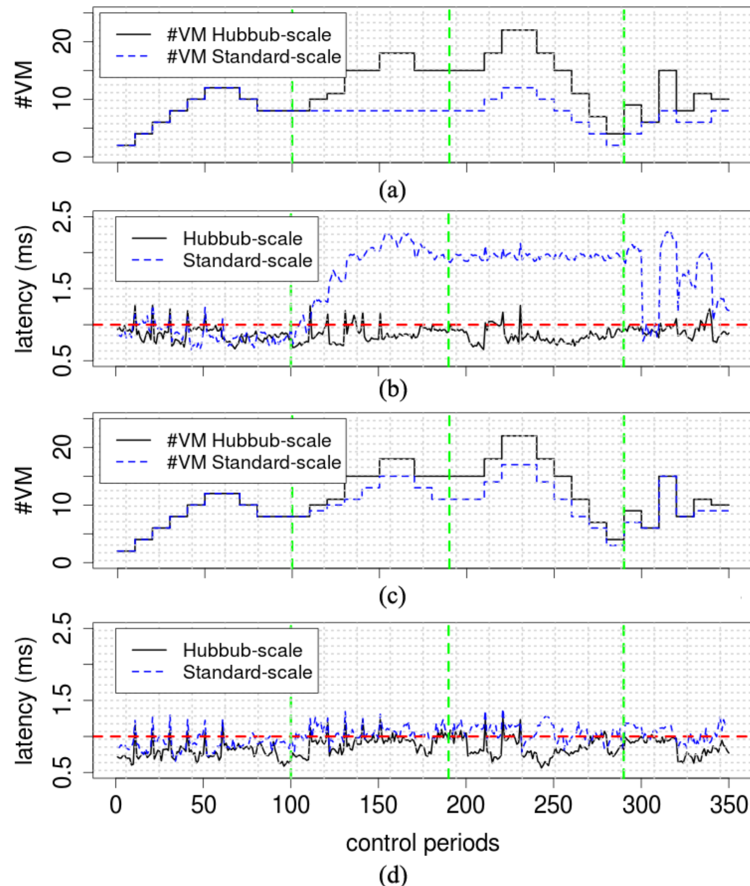


Figure 5.15: Results of running Redis across the different phases. (a) and (b) shows the number of VMs and latency of Redis for a workload based model. (c) and (d) hows the number of VMs and latency of Redis for a CPU based model.

and shows that Hubbub-Scale is aware of interference and spawns enough instances to satisfy the SLO. In section 5.8.3 we show that Hubbub-scale does not over-provision instances to maintain the SLO.

5.8.2.2 Scaling Out using a CPU based Model with/without Interference

We construct a control model based on CPU as explained in section 5. Figure 5.14(d) and 5.15(d) plots the results from scaling out Memcached and Redis during the four different phases. Figure 5.14(c) and 5.15(c) plots the number of active VM instances as the workload intensity and interference intensity

changes in four phases. Hubbub-Scale is aware of interference and adapts to it by spawning the right number of instances. Both Hubbub-Scale and the standard scaling perform equally well during the first phase and provision the same number of VMs to deal with the increasing workload in the absence of any interference. Hubbub-Scale adapts to the increasing interference and spawns more VMs to maintain the SLO requirement while standard modelling approaches fail. Figure 5.13(c) shows the interference index captured during the runtime. Despite running a mix of different interfering applications, the index retains relative meaning and is robust enough to capture the pressure on the memory subsystem.

Our experiments indicate that a standard control model fails to capture the correlation between workload and latency in a multi-tenant scenario. Even a coarse approximation of resource contention is enough to drive the accuracy of the controller by a significant scale and minimize SLO violations.

5.8.3 Utility Measure

An efficient elasticity controller must be able to achieve high resource utilization and at the same time guarantee SLO commitments. Since achieving low latency and high resource utilization are contradictory goals, the utility measure needs to capture the goodness in achieving both these properties. While a system can outperform another in any one of these properties, a fair comparison between different systems can be drawn only when both the aspects are taken into account in composition. To this order, we define the utility measure as the cost incurred:

$$U = VM_hours + Penalty$$

$$Penalty = DurationOfSLAViolations * penalty_factor$$

$DurationOfSLAViolations$ is the duration through the period of the experiment the SLA is violated. We vary the penalty factor which captures the different cost incurred for SLO violations. Figure 5.16 shows the utility measure for 5 different scaling approaches. Ideal scaling represents the theoretical best scaling possible with right VM allocation and no SLO violations. Without any penalty for SLO violations, standard modelling incurs the lowest cost because it allocates only a few instances but results in SLO violations. But as the penalty for SLO violations increase, Hubbub-Scale achieves low utility (cost), which is much better than both standard scaling methods and comparable to

ideal approach. Results with penalty=0 also shows that Hubbub-scale allocates a comparable number of VMs to ideal approach and does not achieve SLO guarantees by unfairly over-provisioning resources. We also note that this is a consequence of the way our experiments are carried out with interference on all physical hosts. With a round robin scheduler, the elasticity controller does over provision to some extent since each host roughly receive the same number of requests, and the maximum requests per server is capped by the lowest amount of workload that can be handled without violating the SLO. This over-provisioning can be mitigated by making the load balancer interference aware.

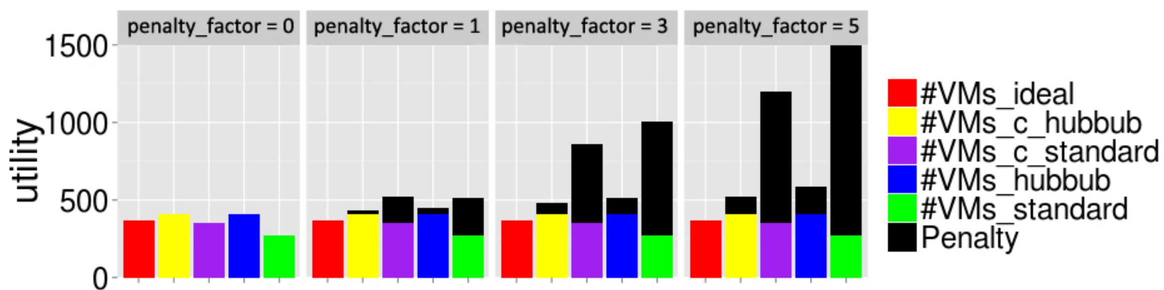


Figure 5.16: Utility measure for different Scaling approaches. VMs_ideal represents the theoretically best scaling possible without any over-provisioning or SLO violations. VMs_c_hubbub and VMs_c_standard represents the utility measure of CPU based scaling using Hubbub and standard modelling respectively. VMs_hubbub and VMs_standard represents the utility measure of workload based scaling using Hubbub and standard modelling respectively.

5.9 Related Work

Elastic Scaling:

There exists a number of work on elastic scaling. In [88], the authors provide a comprehensive view of all the elastic scaling techniques developed. In this section, we highlight the most relevant techniques.

Amazon Auto Scaling [72] is an existing production cloud system which depends on the user to define thresholds for scaling up/down resources. However, it is difficult for the user to know the right scaling conditions. Rightscale [73] is an industrial elastic scaling mechanism and uses load-based threshold to

automatically trigger creation of new virtual instances. It uses an additive-increase controller and can take a long time to converge and know the requisite amount of machines for handling the increasing load.

Reinforcement learning is usually used to understand the application behaviors by building empirical models either online or offline. Simon [74] presents an elasticity controller that integrates several empirical models and switches among them to obtain better performance predictions. The elasticity controller built in [75] uses analytical modeling and machine-learning. They argued that by combining both approaches, it results in better controller accuracy. Although reinforcement-learning mechanisms converge to an optimal policy after a relatively long time, it reward mechanisms cannot adapt to rapidly changing interference as it is unaware of the amount of contention in the system.

Control theory aims to define either a proactive or a reactive controller to automatically adjust the resources based on application demands. Previous works [76, 48, 45, 77] have extensively studied applying control theory to achieve fine grained resource allocations that conform to a given SLO. However, the existing approaches are unaware of interference and will consequently fail to meet the SLO.

In Time series based approach, a given performance metric is sampled periodically at fixed intervals and analysed to make future predictions. Typically these techniques are employed for workload or resource usage prediction and is used to derive a suitable scaling action plan. Chandra et al.[89] perform workload prediction using a histogram and auto-regression methods. Gmach et al.[90] used a Fourier transform-based scheme to perform offline extraction of long-term cyclic workload patterns. PRESS [43] and CloudScale [44] perform long-term cyclic pattern extraction and resource demand prediction to scale up. Although these approaches account for performance interference inherently, they are known to perform well only when periodic patterns exist, which is not always true in a dynamic environment such as Cloud. Our proposed approach using a control model, combines both online and offline training to achieve efficient scaling plans.

5.10 Summary

We conducted systematic experiments to understand the impact of performance interference on CPU utilization and workload, two widely used metrics in elastic scaling. Our observations show that metrics become unreliable and do not accurately reflect the measure of service quality in the face of performance interference. Discounting the number of VMs in a physical host and the amount of interference generated can lead to inefficient scaling decisions that result in under-provisioning or over-provisioning of resources. It becomes imperative to be aware of interference to facilitate accurate scaling decisions. The implication of this observation introduces significant challenges in answering the following questions under multi-tenancy scenarios on: when to scale, how many VMs to launch, and where to place VMs.

We model and quantify performance interference as an index that can be used in the models of elasticity controllers. We demonstrate the usage of this index with CPU utilization and workload intensity by building Hubbub-scale, an elasticity controller that can reliably make scaling decisions in the presence of interference.

Chapter **6**

Augmenting Elastic Scaling for improved accuracy

Abstract

Elastic resource provisioning is used to guarantee service level objective (SLO) with reduced cost in a Cloud platform. However, performance interference in the hosting platform introduces uncertainty in the performance guarantees of provisioned services. In this chapter, we show that assuming predictable performance of VMs in a multi-tenant environment to scale, will result in long periods of QoS degradations. We augment the elasticity controller to be aware of interference and improve the performance in one of three ways: interference aware load-balancing, reduced convergence time when scaling out and informed scaling down. We perform experiments with Memcached and compare our solution against a baseline elasticity controller that is unaware of performance interference. Our results show that augmentation can significantly reduce SLO violations and also save provisioning costs compared to an interference oblivious controller.

6.1 Introduction

With the rise in web services, application and content have moved from being static to more dynamic. This includes social media and networking services that see massive growth in the amount of community generated content. The shift to dynamic content along with increased traffic puts a strain on the sites providing these services. This has resulted in the evolution of caching systems such as Memcached [80], that runs in-memory and act as a caching layer to deliver content at low latency. For example, popular content that is frequently accessed can be replicated in these caching layers to provide low latency access to multiple users. These caching layers must be able to adapt to the varying loads in traffic in order to save provisioning costs and to deliver content at high speeds. Cloud providers such as Amazon provide support for elastic scaling of resources to meet the dynamic demands in traffic. Previous works have proposed multiple models ranging from simple threshold based scaling [72, 73] to complex models based on reinforcement learning [74, 75], control modelling [76, 48, 45, 77], and time series analysis [44, 43] to drive elastic provisioning of resources. The major challenge is to decide when to add/remove resources and to decide the right amount of resources to provision. These challenges are further exacerbated by performance variability issues that are specific to cloud such as performance interference. None of the previous work consider performance variability from interference. Existing research shows that interference is a frequent occurrence in large scale data centers [36]. Therefore, web services hosted in the cloud must be aware of such issues and adapt when needed.

Performance interference happens when behavior of one VM adversely affect the performance of another due to contention in the use of shared resources such as memory bandwidth, last level cache etc. Prior work indicate that, despite having (arguably the best of) schedulers, the public cloud service, Amazon Web Service, shows significant amount of interference [12, 7, 91]. Existing solutions primarily mitigate performance interference and guarantee performance in either one or a combination of these 3 ways: (i) Hardware partitioning (ii) at the scheduling level or (iii) by dynamic reconfiguration. Hardware partitioning techniques involve partitioning the shared resources to provide isolated access to the VMs [13, 14, 15]. They may not be feasible for the existing systems and requires changes to the hardware design. Scheduling

mechanisms look at ways to place together those VMs or threads that do not contend for the same shared resource, essentially minimising the impact of contention [28, 29]. This knowledge is typically accrued through static profiling of the applications. Dynamic reconfiguration involves techniques such as throttling best effort applications or live VM migration upon detecting interference [92]. Reconfiguration techniques such as VM migration involves a huge overhead and throttling best effort applications are possible only if they are co-located on the same host. All these solutions, look at ways to guarantee performance in a multi-tenant setting by either resorting to VM placement or reconfiguring the VMs. Elastic scaling in a cloud environment does not come with the convenience of choosing where to spawn a new VM.

We show that when elasticity controllers are unaware of interference, it either results in long periods of unmet capacity that manifest as Service Level Objective (SLO) violations or results in higher costs from over provisioning. We augment them to be aware of interference to significantly reduce SLO violations and save provisioning costs. We consider Memcached for elastic scaling, as it is widely used as a caching layer, and present a practical solution to augment existing elasticity controllers in 3 ways: (i) At the ingress point by load balancer reconfiguration (ii) by reducing the convergence time when scaling out and (iii) by taking informed decisions when scaling down/removing instances. We achieve this with the help of hardware performance counters to quantify the intensity of interference on the host. We do not rely just on the counters of the target application as they are insufficient to detect interference. We take into account the behaviour of the co-running VMs to quantify interference. This is achieved with the help of a middleware that exposes an API for VMs to query the amount of interference in the host. Decisions by the elasticity controller is then augmented with the help of this information. Our main contributions are:

1. We show that the maximum workload any VM can serve within a SLO constraint is severely impacted by interference. An immediate consequence of this impact is an increase in the time taken for the scaling out process to converge which results in increased SLO violations. We also show that this resulting period of SLO violation is directly proportional to the time taken to spawn and prepare VMs. Our tests on Amazon Web Service (AWS) show that preparing VMs take anywhere between 2 mins to 28 mins depending on the size of data to be transferred.

2. We design and develop a solution to augment elasticity controllers to be aware of interference. Our solution quantifies the capacity of a VM based on the interference experienced on the host by modelling the performance of the target application. With this we are able to reduce the impact of interference on SLO violations by reconfiguring the load balancer, reducing the convergence time when scaling out and by removing highly interfered instances from the cluster when scaling down.

3. We perform experiments with Memcached and compare our solution against a baseline elasticity controller that is unaware of performance interference. We find that with augmentation we can reduce SLO violations by 65% and also save provisioning costs compared to an interference oblivious controller.

6.2 Problem Definition

Any scaling process involves 2 steps: making a decision on when to scale and choosing the right number of instances to be added/removed to serve the changing workload. Both these steps depend on the capacity of a VM. *Capacity* is defined as the maximum amount of workload a VM can serve within a certain level of QoS. A decision to scale-out is established when the elasticity controller detects that the workload exceeds the current capacity of the VMs. It then determines the additional capacity needed to serve the excess workload and spawns the required number of VMs. The accuracy of scaling thus depends on the agility of the controller in detecting workload changes and satisfying the additional capacity. Although modelling techniques [50, 51, 48, 45] can help determine the required capacity, it is difficult to identify the right number of VM instances required to meet the new workload demand. This is because, the capacity of a VM is not determined by the resource specification of a VM alone. Performance interference also impacts the VM capacity. We demonstrate the consequences of this problem through a representative example shown in figure 6.1.

Convergence of Scaling: Convergence time of scaling is the time it takes an elasticity controller to reach a stable desirable state. Figure 6.1 demonstrates the undue delay in convergence of a scaling process because of unmet capacity. From time 0-50 secs, the cluster is able to handle the workload and latency remains below the SLO. After 50 secs the workload increases and the elasticity controller detects the need for additional capacity to serve the increased

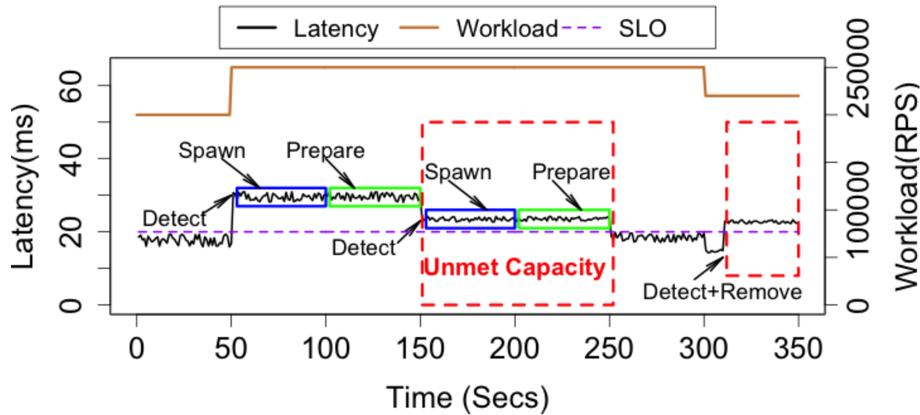


Figure 6.1: Memcached service experiencing a long period of SLO violation during periods or scaling out and scaling down because of unmet capacity from performance interference.

workload. It then spawns the required number of VM instances and prepares the instance with the necessary data to serve the additional workload. The number of instances spawned to serve the additional workload is based on its knowledge of additional capacity needed. We call this the first phase of scaling. Although latency reduces after the first phase of scaling, it still violates the SLO. The new instance (VM_{inter}) happened to be spawned on a server highly impacted by interference. As a result, the VM does not have enough capacity to serve the additional workload within the SLO. This is because performance interference reduces VM capacity. More details on how interference affects capacity is explained in section 6.3.1. Had there been no interference, the SLO would have been maintained after the first phase of scaling. The elasticity controller is unaware of this interference and detects SLO violations at 150 secs and immediately spawns and prepares another instance to maintain the SLO. The period between 150 to 250 secs is the period of unmet capacity from interference and increases the convergence time of the elasticity controller. This period is directly proportional to the time it takes to spawn and prepare a new instance. It results in SLO violations and can only be discovered by the controller after the first phase of scaling. This is because, the elasticity controller is oblivious to interference and cannot know the capacity of the VM before it starts serving the workload. Once another VM is spawned, the SLO

is maintained. The cluster converges to a desirable state only after spawning and preparing this additional VM.

Scaling Down: In the same figure (figure 6.1), we see that the workload drops at around 300 seconds and the elasticity controller detects and removes an instance based on its model of capacity. However, removing the instance immediately results in SLO violations. This is because the controller is unaware of interference and randomly chose a VM to remove and it so happened to be a VM with high capacity (least interfered). The excess workload from removing this VM overwhelms VM_{inter} and exceeds the capacity VM_{inter} can handle. With augmentation, we make informed decisions on choosing which VM to remove.

6.3 Experimental Analysis

We perform experiments for studying the impact of interference on load and VM capacity on a private cloud testbed. It comprises of Intel Xeon X5660 nodes, each with 6 physical cores operating at 2.8GhZ and 12MB of L3 cache. Note that the hardware specification of the physical machine is different from the ones used in the previous chapter. We focus primarily on the performance of in-memory storage systems and run experiments using Memcached to study the impact of interference. Interference is generated using a slew of realistic applications from SPEC CPU benchmark [68] and benchmarks such as mbw [93] and Stream [94]. The experiment to study the time taken to prepare a VM is carried out on AWS, using 2 large instances. We chose to do the experiments of interference on our private testbed because of the difficulty in co-locating VM instances on the same host in AWS.

6.3.1 Interference reduces VM Capacity

In this experiment, we generate different amounts of interference on the memory subsystem when Memcached is serving workload. We set our SLO at 1.8ms and workout the maximum workload that can be handled just within this SLO, for different amounts of interference. The capacity of a VM, i.e., the maximum amount of workload it can handle without violating the SLO is inversely proportional to the amount of performance interference in the host. Figure 6.2 shows this behaviour. The VM has the highest capacity when

running in isolation. As the amount of interference in the host increases, the capacity diminishes. This means that an elasticity controller that is unaware of performance interference will always spawn VMs assuming a much higher capacity than achievable (in the presence of interference) leading to longer periods of SLO violations. With 5x interference the capacity diminishes by upto 35%. This means that, even if an elasticity controller over provisions every VM assuming the capacity at 5x interference, we end up effectively paying an unnecessary cost of 1 VM for every 3 VMs spawned (33% higher cost). We therefore need to augment the elasticity controller with knowledge of interference to minimize provisioning cost and SLO violations.

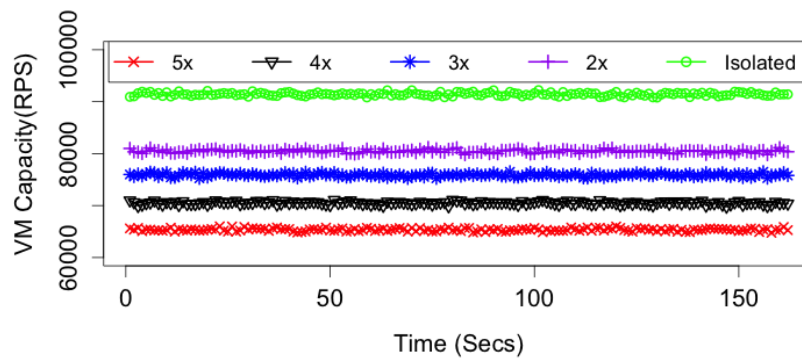


Figure 6.2: VM capacity reduces with increasing performance interference. In this example the SLO is set to 1.8 ms.

6.3.2 Interference vs. Load

In this experiment, we run a standalone instance of Memcached at different loads for a fixed amount of interference. Interference has a significant impact only at high loads. Figure6.3a shows the results from the experiment. Each data point in the figure is an average of a 10 minute run. The impact of interference is negligible until 55000 requests per second and they overlap with each other but as the load increases (> 55000) latency begins to rise sharply in the presence of interference. This observation implies that the impact of interference can be mitigated to a large extent by reducing the workload served on impacted VMs. For example, the load balancer can be configured with weights corresponding to the interference to minimize the impact of interference.

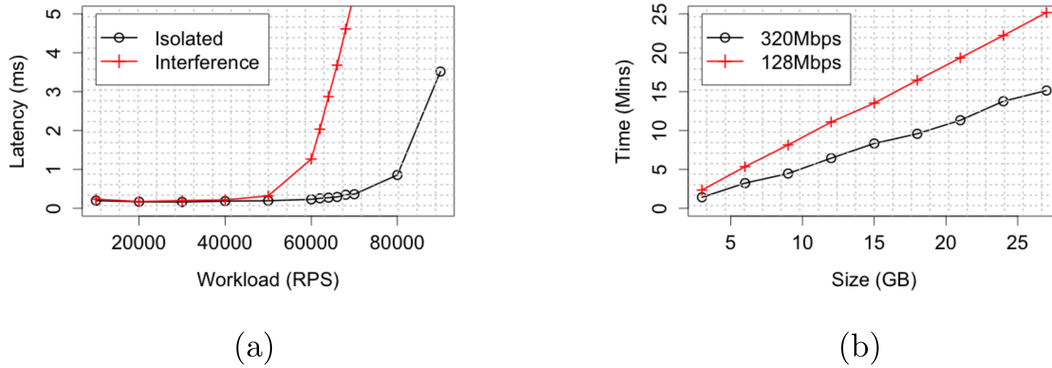


Figure 6.3: (a) Impact of interference for different load on Memcached. Interference impacts performance only at higher loads. (b) Time taken to load Memcached data on a AWS large instance for different data sizes.

6.3.3 Preparation Time

Recall from the previous section that the period of unmet capacity directly depends on the time taken to spawn and prepare a VM instance. We found that the time taken to spawn a VM on AWS takes between 1 to 2 minutes. In this experiment, we evaluate if the time taken to prepare the instance with necessary data is long enough to be a significant problem. Our results from AWS are shown in figure 6.3b. For a caching layer, data loading is done either by the backend server or by any of the other caching instance. This process of data migration by itself affects the performance of the cluster. We loaded data at 2 different data rates, one at full capacity of the data generator and another at a throttled rate. We find that the time taken to load is significant enough and is in the order of minutes. It takes anywhere between 2 to 28 minutes which is significant enough to be considered a problem when met with SLO violations for that period. In the next section we present the design and implementation of the approach to augment elasticity controllers for improved accuracy.

6.4 Solution Overview

Our solution primarily consists of 2 components: i) An Augmentation Engine (AE) and ii) a Middleware Interface to quantify capacity (MI). MI resides on

all the hosts in the cluster and exposes an API that can quantify the maximum amount of workload a VM on the host can handle without violating the SLO. MI computes this based on the amount of contention in the shared resources of the host. The Augmentation Engine orchestrates with the MI to make scaling decisions. The architecture of the system components are shown in figure6.4

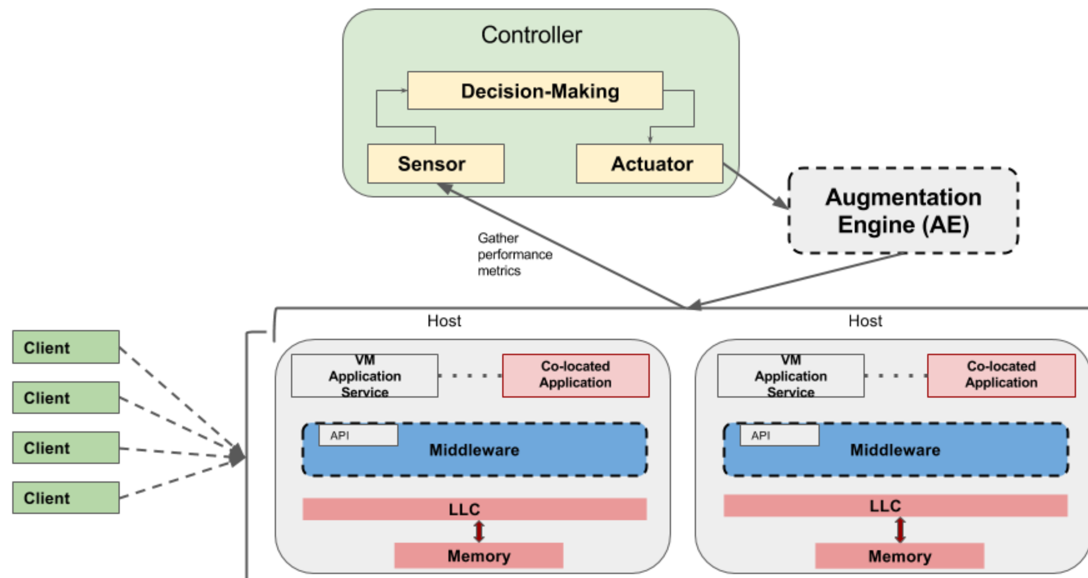


Figure 6.4: Architecture of the system components

First the AE receives the decision from the elasticity controller (EC) to scale out/scale down. It also receives the number of instances to add/remove respectively. In essence, the AE acts as a delegator to act on the decisions of the EC. Before acting on the input, the AE consults the MI on all hosts to quantify the capacity of each host. AE takes a decision by itself only when it has enough historical evidence to believe that the interference is sustained. Based on this knowledge of host capacity, along with the current workload, AE first tries to generate a plan to balance the load among hosts without adding new VM instances. The plan aims to reduce the load on highly interfered VMs by diverting the workload to VMs that are less interfered. If such a plan is feasible, then the decision from the EC is overruled and the AE reconfigures the load balancer. This is the first level of reconfiguration. However, such a plan may not always be possible. If the AE learns that a rebalancing solution alone cannot maintain the SLO, it then directly acts on the decision of the EC. If the

decision of the EC is to remove instances, it consults the MI, recomputes the overall capacity needed to serve the workload and removes the right number of highly interfered instances. If the decision is to add new instances, the AE adds only so many instances as directed by the EC. It then waits for the new VMs to spawn. As soon as they are spawned, the MI on the hosts of the newly spawned VMs are consulted to learn the capacity of these VMs. Note that the MI need not wait for the VMs to finish preparing the instance to know the capacity. This is because the only hardware performance counter of Memcached that the MI relies on, to quantify interference, is the cache-reference rate (for other counters used from co-running VMs, see section 6.5 for more details) and this is unaffected by the hit rate of requests on Memcached. This counter only captures the effective rate at which the cache is accessed, which is only dependent on the rate at which the instance receives the requests. The preparation phase is briefly paused for a few seconds to determine the capacity based on the current level of interference. It is precisely because of this capability that the AE can determine if the newly spawned instances are enough to handle the increased workload even before the preparation phase is fully complete. If the newly added instances are incapable of maintaining the SLO, the AE spawns additional instances in parallel with the prepare phase of the previous instantiation. This process of parallel instantiation along side the prepare phase significantly reduces the duration of SLO violations.

The results from augmenting the elasticity controller in figure 6.1 is shown in figure 6.5. Note how the periods of unmet capacity are significantly reduced. Spawning and preparing VMs in parallel with the preparation of the VMs in the first phase of scaling minimises SLO violations. Similarly, when scaling down, there are no SLO violations since the augmented approach is aware of interference and removes the VM that is highly interfered (VM_{inter}).

6.5 Middleware Interface to Quantify Capacity (MI)

The middleware interface (MI) exposes an API that is responsible for quantifying the capacity of the VM in terms of the maximum workload it can handle for a given SLO. In order to quantify the capacity, MI first needs to quantify the amount of interference in the host. The primary role of MI is to quantify the drop in the performance of target application from interference

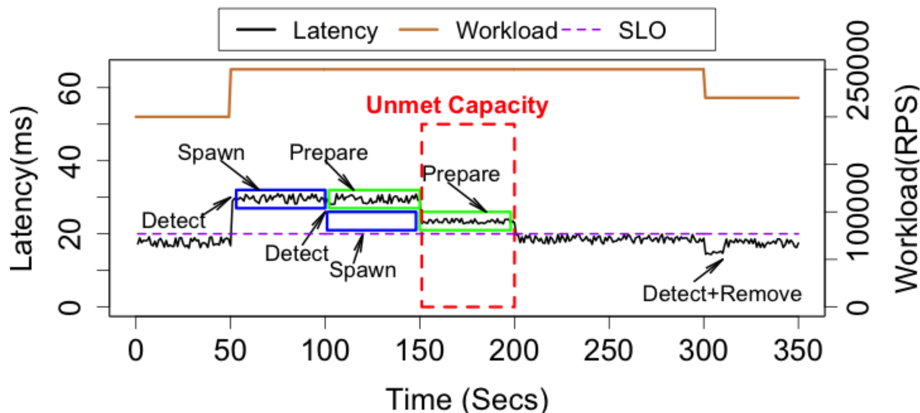


Figure 6.5: Unmet capacity from figure 6.1 pruned by augmenting the elasticity controller to detect and quantify interference.

and to translate this performance degradation to capacity. We use the same middleware designed and explained in chapter 5.

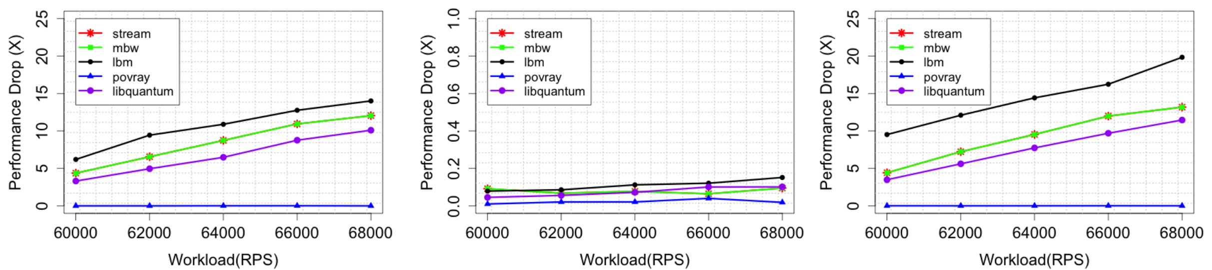
6.5.1 Characterising Contention:

In order to characterise contention, we choose in-memory storage system Memcached as a demonstrative target system to show the scaling of services under performance interference.

We say that the storage system *co-runs* with other applications when they all run on different cores of the same physical host; we refer to all these applications as *co-runners*. In our experiments, we compute the performance drop as follows: First, we measure the average latency L_i of the storage system when running in isolation. Then we measure the average latency L_c of the storage system when it co-runs with other processes. Performance drop suffered is $\frac{L_c - L_i}{L_i}$.

Sources of degradation: There are 2 main subsystems responsible for contention: the cache and the memory bandwidth. In order to assess the impact of contention on these subsystems, we use different system configurations that are designed to generate contention at different resources: the first configuration generates contention only on the cache, the second only on the memory bandwidth and the third one on both. Figure 6.6 show the drop

in performance experienced by Memcached for all the three configurations. It is clear that cache is the dominant source of performance degradation, contributing upto 15X drop in performance and bandwidth contributing less than 1X. The results show that Memcached benefits more from it's reliance on the cache than from memory bandwidth. What is also interesting to note is that the drop in latency increases linearly for the same co-runner. We only plot the characterisation of Memcached 60000 RPS and up, since, Memcached remains unaffected by interference below that workload (see figure 6.3a).



(a) Contention for L3 cache (b) Contention for memory bandwidth (c) Contention for both resources

Figure 6.6: The drop in performance of Memcached for contention at different levels of memory subsystem. Memcached is run alongside multiple instances of different co-runners.

Properties that determine degradation: We investigate properties of the co-running application that cause performance degradation. In figure 6.6, all the different co-runners cause degradation in the same order; ie. lbm consistently causes the highest amount of performance degradation, followed by mbw, stream, and povray. Note that this is different from the order observed in the previous chapter. This difference arises from the heterogeneity in the hardware used for the experiments. In order to understand the properties that define the aggressiveness of the co-runners, we rely on PMUs. Since cache is the dominant source of degradation, cache references and cache misses of the co-runner is a good indicator of cache contention. Intutively, higher cache references from the co-runners effectively reduces the cache space of Memcached, resulting in a drop in performance. However, this alone is insufficient to model the sensitivity of Memcached. Upon deeper analysis, we find that the order of co-runners that cause performance drop

(lbm>mbw=stream>libquantum>povray) does not correspond with their cache access rate. This is because cache access rate alone does not take into account the memory access patterns of the co-runners. LLC prefetches and LLC prefetch misses of the co-running application gives an approximation of the memory access patterns. Finally, the extent of degradation suffered by cache access reduction is captured by cache-reference counter for Memcached. The higher the degradation suffered, the lesser the cache-reference counter for Memcached. From our experiments, we find that these counters provide a good model to quantify sensitivity of Memcached. The final list of performance counters chosen to quantify sensitivity are shown in table 6.1.

Interference-Index: The goal of characterising contention is to quantify the properties of the co-runners that lead to performance degradation of the storage system. We call this metric interference-index and it approximates the performance degradation suffered by the storage system. In order to be useful, the metric must correlate with the performance drop suffered by the storage system.

Name	**Description**	**Name**	**Description**
cpu-clk	Reference cycles	inst-retired	Instructions retired
cache-ref	References to L3 cache	cache-miss	L3 cache misses
llc-prefetch	L3 prefetches	llc-prefetch-miss	L3 prefetch misses

Table 6.1: Performance counters included in characterising contention

We derive a set of N representative performance counters (shown in table 6.1) $WS = m_1, m_2, ..., m_N$ where m_i represents the metric i. Using these counters and a training data set of co-runners, our system then builds a model that correlates co-runner properties with performance drop suffered by the storage system. We then use linear regression on these counters to construct the interference index. Figure 6.7 shows the interference-index constructed for Memcached. Since the modeling is data-driven, the interference index generated is application-dependent. We however do not view this as an issue since modeling can be fully automated. From figure 6.7, we see that

interference-index correlates with performance drop suffered by Memcached. Higher the interference-index, greater the performance drop suffered. In this example, our training set consists of lbm,mbw,libquantum and povray. We then fit stream, milc and omnetpp into the generated model. Stream causes similar degradation as mbw and they both correspond to similar interference indexes. milc causes a degradation that is greater than povray and omnetpp but lesser than mbw and is also captured by the model as expected. The value of interference index approximately corresponds to the drop in performance of Memcached. Once interference-index is quantified, it is then used as a control input to determine the capacity of the VM. Note that because of the linear trend in performance drop above 60000 RPS (figure 6.6), knowing a workload and interference index, it is possible to estimate the drop in latency for any other workload.

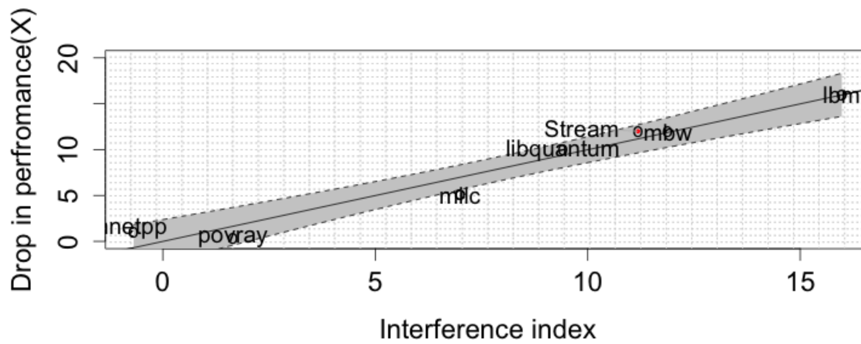


Figure 6.7: Interference-index quantifies the performance drop suffered by Memcached based on the behaviour of the co-runner.

Capacity: We apply a binary classifier used in the state of the art approaches [50, 51] to approximate the capacity of a VM, which defines its ability to handle workloads under the SLO. In state of the art approaches, models are built by finding correlations between Monitored Parameters (MP), which reflects the operating state of the system, and the Target Parameter (TP), in which the SLO is defined. The MPs are chosen by analyzing whether they have remarkable influences on the SLO. Previous works adopt CPU utilization, workload intensity and workload composition, including read/write ratio, data size as typical MPs. In addition, our model includes the interference index, which, as shown in previous sections, significantly influences the SLO. With sufficient training data, the classifier uses SVM (support vector machine) to

estimate a function, which outputs satisfying (true) or not satisfying (false) the SLO by giving the MPs and the constraint of the TP. Then, we are able to obtain the maximum supported workload intensity under the current MPs while satisfying the SLO. The maximum supported workload intensity defines the capacity of a VM.

The model used is trained offline using the training data collected when running Memcached under different intensities of the interference and workload. Various kinds of workload, including varying compositions and RPS, are generated for Memcached. Multiple intensities of interference and workload provide sufficient variance in the interference index, which covers most of the operating space of the model. With little modifications, the model can also be adapted and automated to get trained online.

Overhead: Our middleware has a very minimal overhead since it only samples the counters every few seconds. It has a negligible CPU consumption of less than 3% and does not perform any instrumentation to the application that result in performance loss. Augmentation incurs very negligible overhead since the only additional step involved is the construction of interference index, which in itself relies on non-intrusive monitoring.

6.6 Augmentation Engine

The AE operates with the estimation of the capacities of VMs using an empirical model explained in Section 6.5. The capacity of a VM defines the ability of the VM to serve workload, in terms of requests per second, under the current workload composition. Without knowing the interference index of a VM, the capacity estimation is the most optimistic and is denoted by $\underline{C}$, which is a function of the current workload composition and the VM flavor ($\underline{C} = f(W_t, F)$). Taking into account of the interference index, the capacity of a VM is represented by C, which is a function of the current workload composition, the VM flavor and the interference index ($C = g(W_t, F, I_t)$). Let L_t denotes the total amount of workload received by the system at time t and each procedure starts with n VMs in the cluster providing $\sum_{i=1}^{n} C_i$ capacity. The procedures during load balancer reconfiguration, scaling out and scaling down are described in Algorithm 1

Load Balancer Reconfiguration: The load balancer reconfiguration procedure is triggered when the AE receives a scaling out decision from the EC or after scaling out/down. Assuming the capacities of each VM in the current cluster setup are $C_1, C_2 \ldots C_n$ and the total capacity provided is $\sum_{i=1}^{n} C_i$. When $\sum_{i=1}^{n} C_i \succeq L_t$, the cluster is able to operate under the SLO using load balancer reconfiguration. Then, the AE reset weights $(W_1, W_2 \ldots W_n)$, that are proportional to the capacity of each VM $(C_1, C_2 \ldots C_n)$, in the load balancer. Otherwise, the AE continues with the scaling out approach.

Scaling out: During scaling out procedure, the AE spawns the number of VMs indicated by the EC. After the VMs are spawned, their interference index are assessed and their capacities $\Delta C_1, \Delta C_2 \ldots \Delta C_m$ under the current workload are estimated. In the meantime, these VMs start to prepare. Whether the current setup with proper load balancing is able to satisfy the current workload L_t under the SLO is evaluated by $\sum_{i=1}^{n} C_i + \sum_{j=1}^{m} \Delta C_j \succeq L_t$. If the above inequality holds, proper weights of both existing and additional VMs are reset in the load balancer and the scaling out procedure exits. Otherwise, the unmet capacity $C_{unmet} = L_t - (\sum_{i=1}^{n} C_i + \sum_{j=1}^{m} \Delta C_j)$ needs to be handled by spawning another batch of VMs, whose capacities are optimistically estimated $(\underline{C_1}, \underline{C_2} \ldots \underline{C_k})$, where $\sum_{i=1}^{k-1} \underline{C_i} \preceq C_{unmet} \preceq \sum_{i=1}^{k} \underline{C_i}$. Then, the AE spawns the corresponding k VMs like executing scaling out commands from the EC. This procedure iterates until $C_{unmet} \preceq 0$.

Scaling down: When the EC issues a scaling down decision, the AE overrules the amount of VMs to be removed issued by the EC and judiciously removes the most interfered VMs. This is because the most interfered VM serves the least amount of workload within the SLO for the same price as a non-interfered VM. The number of VMs to be removed by the AE satisfies $\sum_{i=1}^{m} \Delta C_i \preceq C_{extra} \preceq \sum_{i=1}^{m+1} \Delta C_i$, where $C_{extra} = \sum_{i=1}^{n} C_i - L_t$. By selecting and removing the highly interfered VMs, the AE usually removes more VMs than the amount issued by the EC, which saves the provisioning cost.

Algorithm 1 Augmentation Engine

```
1: procedure LoadBalancerReconfiguration()
2:     if ∑_{i=1}^{n} C_i ⪰ L_t then
3:                                   ▷ Calculate capacity of VMs C_1, C_2 ... C_n
4:         with C_i = g(W_t, F, I_t)
5:                                   ▷ Reset weights W_1, W_2 ... W_n
6:         with W_i = L_t / (∑_{j=1}^{n} C_j) * C_i
7:     else
8:         ScalingOut()
9: procedure ScalingOut(m)
10:    Spawn m VMs as indicated by the EC
11:    Prepare these m VMs immediately and concurrently
12:                                  ▷ Calculate extra capacity ∑_{j=1}^{m} ΔC_j
13:    with C_i = g(W_t, F, I_t)
14:                                  ▷ Calculate unmet capacity
15:    with C_unmet = L_t − (∑_{i=1}^{n} C_i + ∑_{j=1}^{m} ΔC_j)
16:    while C_unmet ⪰ 0 do
17:        Spawn extra k VMs
18:        with ∑_{i=1}^{k−1} C_i ⪯ C_unmet ⪯ ∑_{i=1}^{k} C_i
19:        Prepare these k VMs immediately and concurrently
20:                                  ▷ Update unmet capacity
21:        with C_unmet = C_unmet − ∑_{p=1}^{k} ΔC_p
22:    LoadBalancerReconfiguration()
23: procedure ScalingDown(m)
24:                                  ▷ Calculate extra capacity
25:    with C_extra = ∑_{i=1}^{n} C_i − L_t
26:                                  ▷ identify VMs to remove
27:    with ∑_{i=1}^{m} ΔC_i ⪯ C_extra ⪯ ∑_{i=1}^{m+1} ΔC_i
28:    where these m VMs are highly interfered
29:    Remove VMs
30:    LoadBalancerReconfiguration()
```

6.7 Experimental Evaluation

6.7.1 Assumptions

In this work we focus on Memcached for elastic scaling. In [95], Nishtala et.al show how they scale Memcached at Facebook and maintain consistency across regions. Our approach is complementary to this work and enables elastic scaling under SLO constraints in a multi-tenant environment. Cache invalidation and consistency is orthogonal to this work and the storage infrastructure is expected to manage the same. We particularly focus on replication and the backend server is responsible for populating and preparing the VMs. As a result the Memcached instances already present in the cluster don't incur any additional overhead of migrating data.

6.7.2 Experiment Setup

Our experiment setup is the same as in section 6.3. We co-locate memory intensive VMs with the storage system on the same socket for varying degrees of interference by adding and removing the number of instances. MBW, Stream and SPEC CPU benchmarks are run in different combinations to generate interference. We use HAproxy for load balancing the request. In all our experiments we disable DVFS from the host OS using the Linux CPU-freq subsystem. Our middleware performs fine-grained monitoring by frequently sampling the CPU utilization and the different performance counters for all the VMs on the host and repeatedly updates the interference index every 10 secs.

Since the design of elasticity controller is not the focus of this work, we build a simplified version of an elasticity controller similar to [45]. The elasticity controller relies on sampled request latency instead of CPU utilization on each VM. Given our evaluation scenarios, we ensure that the elasticity controller is able to make ideal scaling decisions when interference is not present.

6.7.3 Results

We design our experiments to highlight the inefficiencies in elastic scaling when performance interference is not taken into account and answer the

following questions: i) How much reduction in SLO violations is achieved with augmentation as compared to standard approach without any augmentation? ii) Can augmentation save cost of the hosting services? Specifically, we demonstrate the benefits of load-balancer reconfiguration, and the role of augmentation in minimising SLO violations and saving provisioning costs in a multi-tenant environment.

Server Timeline: Each experiment has a server timeline that is depicted to explain the following status of the servers (i) Whether a Memcached instance is running on the server. White indicates that there is no instance on the server. (ii) Blue indicates that the instance on the server is preparing the data and (iii) Gradient of red indicates the amount of interference on the server. Dark red indicates high interference. The gradient of interference intensity is plotted according to the interference index computed on each server. S1, S2 and S3 indicate the different servers used in the experiment. The server timeline graph is presented for both approaches; scaling with augmentation and without augmentation, which we call the standard/baseline approach. The server timeline essentially aids in understanding and reasoning about the choices made from augmentation and helps compare the cost involved.

6.7.3.1 Load Balancer Reconfiguration

Our first experiment is focused at showing the advantages of load-balancer reconfiguration when compared to a statically configured cluster (i.e. all the nodes in the cluster receive equal number of requests). In this experiment, the cluster receives a constant workload and interference is introduced around 120 secs in one of the servers. The results of the experiment is shown in Figure 6.8. Latency vs. time shows the comparison of latency when the cluster is augmented to reconfigure the load balancer against the standard approach (baseline) with no reconfiguration. The standard approach without augmentation experiences long periods of SLO violation. The same figure also depicts a server timeline of all the servers (shown as S1,S2 and S3) to visualise the server status.

Looking at the standard server timeline, we see that Memcached instances are running only on S1 and S2 initially and they are able to serve the workload without violating the SLO until 120 seconds. The latency remains far below the SLO before 120 seconds. We note that the servers are not over provisioned. The

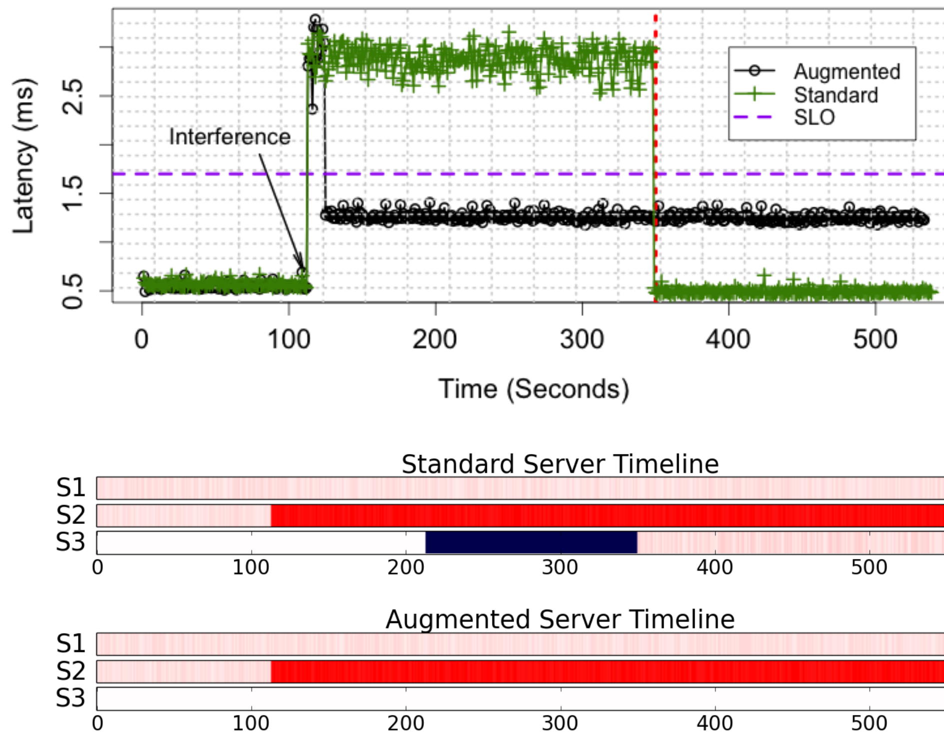


Figure 6.8: Results demonstrating load balancer reconfiguration by AE. Server timeline shows the status of the server over time. White indicates that there is no instance on the server. Blue indicates that the instance on the server is preparing the data. Gradient of red indicates the amount of interference on the server. Dark red indicates high interference

latency stays far below the SLO because of the granularity of VM specification. The workload generated at these 2 machines cannot be served within the SLO by one server alone and having a additional server drops the latency far below the SLO. At 120 seconds, S2 experiences a high amount of interference that causes a spike in latency resulting in SLO violations. As soon as sustained SLO violations are detected, the standard approach spawns a new instance on S3 (not shown in the server timeline, as nothing resides on S3 to show spawning) and just after 200 seconds begins to prepare the instance on S3. This instance cannot serve any workload during the preparation phase and the SLO continues to remain violated. The preparation phase finishes around 350 secs and the Memcached instance on S3 is up and running and begins to serve the workload thereby maintaining the SLO.

The augmented approach behaves differently when compared to the standard approach. After around 120 seconds, when the augmentation engine (AE) detects sustained SLO violations, it first tries to generate a plan for load balancer reconfiguration. AE learns that there is excess capacity on S1 and reroutes the traffic to reduce the load on S2 which is highly interfered. We found that AE routed 60% of the workload to S2 and 40% to S1. This reconfiguration was enough to maintain the SLO. From the augmented server timeline in figure 6.8, we see that no Memcached instance was instantiated on S3. With augmentation, the AE was able to avoid instantiating a new instance, while minimising SLO violations without any additional cost.

6.7.3.2 Convergence when scaling out

Next, we highlight the problem of unmet capacity as a result of delayed convergence. Our experiment shows how augmentation converges quicker in the presence of performance interference while scaling out. In this experiment the total workload served to the cluster is increased so that it has to scale out to maintain the SLO. Figure 6.9 shows the results from the experiment. The workload increases at around 100 seconds and the SLO is violated. The elasticity controller decides to spawn a new VM to handle the increased workload. The standard server timeline in the same figure shows the status of the server. Only S1 was serving the requests initially, but when the workload increases, a new instance is spawned and prepared on S2. S2 is very highly interfered and although the latency dropped, it is still unable to meet the workload demands within the SLO. The elasticity controller learns this only after the instance is prepared and begins to serve the requests. Another instance is then spawned and prepared on S3 and the SLO is maintained.

Augmented approach functions similar to the standard approach up until the point of spawning S2. This can be seen from the augmented server timeline in figure 6.9. But as soon as S2 is spawned, AE learns that S2 is very highly interfered and does not have enough capacity to serve the workload within the SLO. In this particular scenario, even a load balancer reconfiguration does not help. So the AE immediately spawns and prepares another instance on S3 in parallel with the preparation phase on S2. By preparing the new instance in parallel, we minimise SLO violations proportional to the preparation time. In this experiment, the preparation time was a little over 3 minutes and SLO violations are significantly reduced.

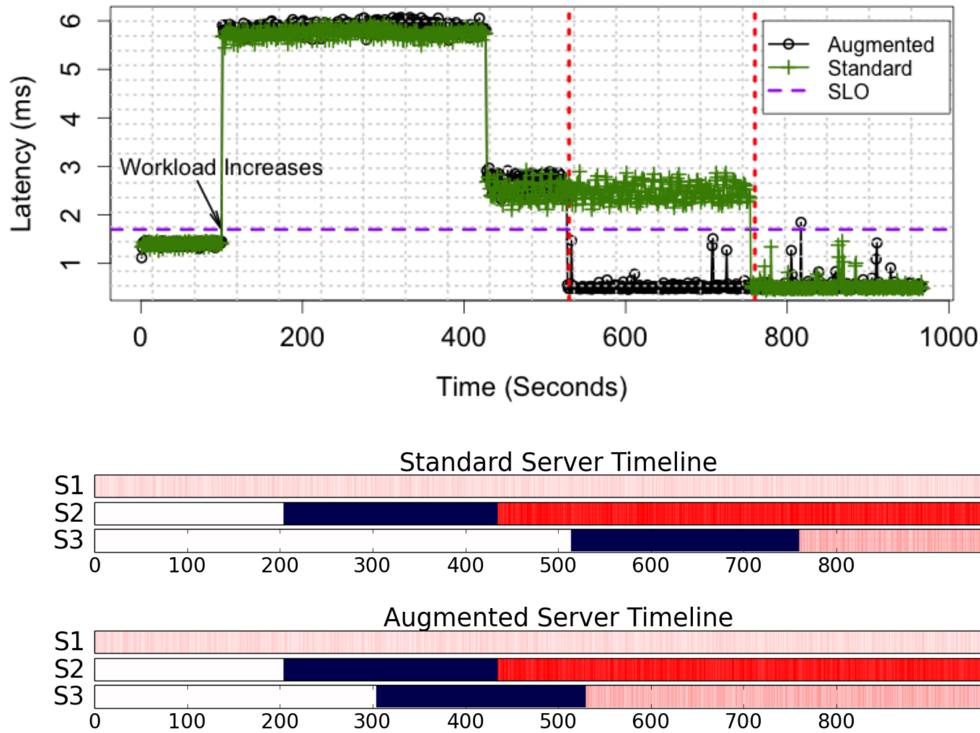


Figure 6.9: Results demonstrating convergence time with AE and without AE. Server timeline shows the status of the server over time. White indicates that there is no instance on the server. Blue indicates that the instance on the server is preparing the data. Gradient of red indicates the amount of interference on the server. Dark red indicates high interference

6.7.3.3 Scaling Down

In this experiment, we highlight how informed decision when scaling down can lower cost and also reduce SLO violations. Initially the cluster has 3 servers running Memcached instances, all of them serving the workload. Around 100 secs later, the workload drops so much that a VM can be removed. In the standard approach without augmentation, the elasticity controller precisely does that when it sees sustained drop in latency and removes a VM. The results are shown in figure 6.10. From the standard server timeline in the same figure we see that the VM on S3 was removed. Since the elasticity controller has no information about the interference on different servers, it randomly chose a server to remove the VM. However, this decision ends up costing more in terms of SLO violations. This is because, the VM on S3 had a high capacity and also

a lot of spare capacity. By removing the VM instance on S3, S2 and S1 have to serve a higher number of requests. Ideally, this should not have caused any SLO violations if all the VMs had the same capacity. But since S2 is highly interfered, it was already serving out on its max capacity and by removing S3, S2 didn't have enough spare capacity to serve the additional workload, resulting in SLO violations. Note that, in the standard approach the cluster is statically configured and all the servers receive equal number of requests. When the controller notices sustained SLO violations, it reinstantiates S3 and the SLO conditions are met.

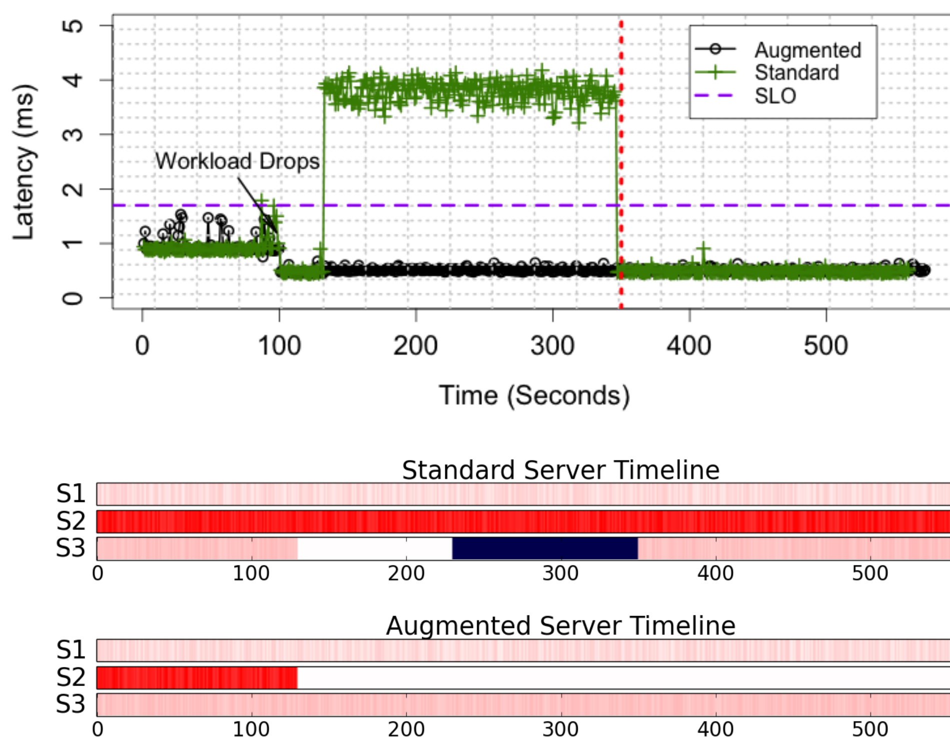


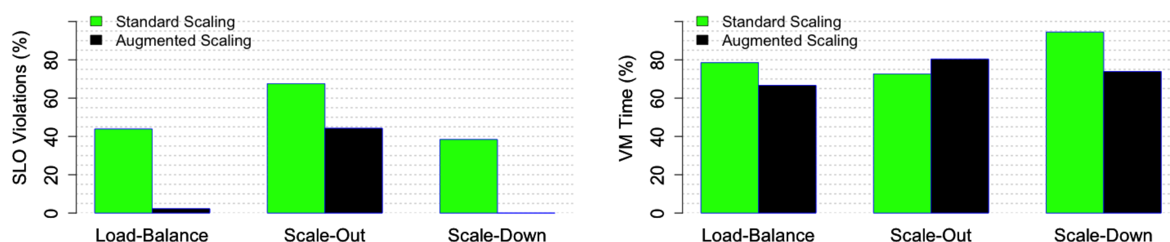
Figure 6.10: Results demonstrating informed scaling down with AE. Server timeline shows the status of the server over time. White indicates that there is no instance on the server. Blue indicates that the instance on the server is preparing the data. Gradient of red indicates the amount of interference on the server. Dark red indicates high interference

By augmenting the elasticity controller, the AE knows the capacity of each VM and upon receiving a decision from the controller to remove a VM, it removes the VM with the least capacity, VM on S2 in this example. Since S2 was not contributing enough to serving the workload, removing S2 doesn't impact

the cluster load significantly and the SLO is maintained. This can be seen in the augmented server timeline in figure 6.10. This experiment demonstrates that augmentation can help make informed decisions when scaling down and remove VM instances with smaller capacity, thereby reducing the cost.

6.7.4 Quantifying gains

An efficient elasticity controller must be able to achieve high resource utilization and at the same time guarantee SLO commitments. Figure 6.11a shows the gains in performance and figure 6.11b shows the corresponding VM time spent. Augmenting an interference oblivious controller not only reduces SLO violations but also saves provisioning cost.



(a) SLO violations suffered for different scenarios

(b) VM time spent for different scenarios

Figure 6.11

6.8 Summary

In this chapter, we show that an elasticity controller cannot make accurate scaling decisions under the interference imposed by co-running applications sharing the infrastructure. It becomes imperative to be aware of interference to facilitate accurate scaling decisions. We design and implement a middleware that augments the decisions made by elasticity controllers. Evaluations have shown that, with our middleware, an elasticity controller is able to experience significantly less SLO violations and provisioning cost under the presence of interference.

Chapter 7

Conclusions

We have investigated the role of performance interference and how it can be mitigated to not only provide performance guarantees but also alleviate the need for undue over-provisioning.

The methods proposed were designed with a specific scenario in mind. The first challenge was to improve the system utilisation in a single host environment where we assume that the workload for the performance sensitive application does not overload the host. We proposed co-locating such performance sensitive applications with best effort batch application to improve utilisation. The batch applications are only executed during periods of low utilisation by the performance sensitive applications. We designed and proposed a methodology to seamlessly and progressively learn the collective behaviour of all the applications in the host to predictively avoid performance interference without the need for any additional instrumentation.

Our next challenge was to address the role of performance interference in a distributed context, where the workload of a performance sensitive application can overload a single host and requires elastic scaling to handle the increased workload. We identified that the typical system metrics used for deciding when to scale become unreliable in a multi-tenant environment and can result in highly inefficient scaling. We proposed a middleware that relies on low level hardware performance counters to quantify the impact of interference and demonstrated that accounting for its role makes the system metrics reliable for elastic scaling decisions.

The last challenge we addressed in this thesis was to reduce the overall provisioning cost and at the same time improve performance during elastic scaling. We identified three major consequences of performance interference that when unaccounted for severely degrades performance and increases cost. We proposed a solution consisting of performance interference aware load-balancing, scaling out, and scaling down that when considered in unison reduces provisioning costs and significantly boosts performance.

We believe that our contribution in whole is a step towards efficient resource management in consolidated environments. The solutions proposed require the service provider to incorporate and expose mechanisms for the users to handle performance interference. Moving complex decisions away from the service provider and towards users allows important management actions –be it managing resource requirements, scheduling tasks– to reside on the users-end who are better aware of the application, while the service provider facilitates this by providing the users with a rich set of information to aid management actions.

Although this thesis is geared towards improving performance in consolidated environments, it does not consider end-to-end performance. End-to-end performance guarantees in a distributed system is not the same as guaranteeing performance for a single component of a system. Depending on the type of application, a component may or may not be in the critical path of the application work flow. The presence of interference further exacerbates the challenges, as it introduces the need for additional knowledge on how the distribution of interference between the different components of the system affect the overall end-to-end performance. To this end, we did some preliminary experiments with Apache spark and our results indicate that different classes of application experience different amounts of degradation depending on the intensity of interference and the distribution of interference. In particular, distributed triangle counting was extremely sensitive to performance interference. Even the presence of interference on a single host, resulted in severe performance degradation overall. On the other hand, application like k-means was more robust and experienced degradation only in the presence of high interference on majority of the physical hosts comprising the application. This knowledge of interference distribution is critical in addressing end-to-end performance guarantees for distributed systems. With the plethora of real time streaming analytics, the need for end-to-end guarantees are becoming increasingly im-

portant and, as we pointed out, addressing this in an efficient manner involves numerous challenges and is part of our future work.

Bibliography

[1] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. Planetlab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

1

[2] Bart Braem, Chris Blondia, Christoph Barz, Henning Rogge, Felix Freitag, Leandro Navarro, Joseph Bonicioli, Stavros Papathanasiou, Pau Escrich, Roger Baig Viñas, et al. A case for research with and on community networks. *ACM SIGCOMM Computer Communication Review*, 43(3):68–73, 2013.

1, 39

[3] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *IEEE computer*, 40(12):33–37, 2007.

3, 15

[4] Linux Nice Utility. `https://en.wikipedia.org/wiki/Nice_(Unix)`.

5

[5] Intel CAT Technology. `http://www.intel.com/content/www/us/en/communications/cache-monitoring-cache-allocation-technologies.html`. accessed: July 2016.

7

[6] Nedeljko Vasić, Dejan Novaković, Svetozar Miučin, Dejan Kostić, and Ricardo Bianchini. Dejavu: accelerating resource allocation in virtualized

environments. *ACM SIGARCH Computer Architecture News*, 40(1):423–436, 2012.

7

[7] Dejan Novakovic, Nedeljko Vasic, Stanko Novakovic, Dejan Kostic, and Ricardo Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. Technical report, 2013.

7, 86

[8] 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*, pages 237–250. ACM, 2010.

8

[9] Lingjia Tang, Jason Mars, and Mary Lou Soffa. Compiling for niceness: Mitigating contention for qos in warehouse scale computers. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 1–12. ACM, 2012.

8

[10] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: improving resource efficiency at scale. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 450–462. ACM, 2015.

8

[11] Fei Xu, Fangming Liu, Linghui Liu, Hai Jin, Bo Li, and Baochun Li. iaware: Making live migration of virtual machines interference-aware in the cloud. *IEEE Transactions on Computers*, 63(12):3012–3025, 2014.

8

[12] Amiya K Maji, Subrata Mitra, Bowen Zhou, Saurabh Bagchi, and Akshat Verma. Mitigating interference in cloud services by middleware reconfiguration. In *Proceedings of the 15th International Middleware Conference*, pages 277–288. ACM, 2014.

8, 12, 86

[13] Fei Guo, Yan Solihin, Li Zhao, and Ravishankar Iyer. A framework for providing quality of service in chip multi-processors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 343–355. IEEE Computer Society, 2007.

9, 86

[14] Miquel Moreto, Francisco J Cazorla, Alex Ramirez, Rizos Sakellariou, and Mateo Valero. Flexdcp: a qos framework for cmp architectures. *ACM SIGOPS Operating Systems Review*, 43(2):86–96, 2009.

9, 86

[15] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. Qos policies and architecture for cache/memory in cmp platforms. In *ACM SIGMETRICS Performance Evaluation Review*, volume 35, pages 25–36. ACM, 2007.

9, 86

[16] Harshad Kasture and Daniel Sanchez. Ubik: efficient cache sharing with strict qos for latency-critical workloads. In *ACM SIGPLAN Notices*, volume 49, pages 729–742. ACM, 2014.

9

[17] Mihai Dobrescu, Katerina Argyraki, and Sylvia Ratnasamy. Toward predictable performance in software packet-processing platforms. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 11–11. USENIX Association, 2012.

9, 67

[18] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 129–142. ACM, 2010.

9

[19] Jacob Machina and Angela Sodan. Predicting cache needs and cache sensitivity for applications in cloud computing on cmp servers with configurable caches. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

9

[20] Younggyun Koh, Rob C Knauerhase, Paul Brett, Mic Bowman, Zhihua Wen, and Calton Pu. An analysis of performance interference effects in virtual environments. In *ISPASS*, pages 200–209, 2007.

9

[21] Andreas Sandberg, Andreas Sembrant, Erik Hagersten, and David Black-Schaffer. Modeling performance variation due to cache sharing. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 155–166. IEEE, 2013.

9

[22] Chi Xu, Xi Chen, Robert P Dick, and Zhuoqing Morley Mao. Cache contention and application performance prediction for multi-core systems. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 76–86. IEEE, 2010.

10

[23] Xi Chen, Chi Xu, Robert P Dick, and Zhuoqing Morley Mao. Performance and power modeling in a multi-programmed multi-core environment. In *Proceedings of the 47th Design Automation Conference*, pages 813–818. ACM, 2010.

10

[24] David Eklov, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. Cache pirating: Measuring the curse of the shared cache. In *2011 International Conference on Parallel Processing*, pages 165–175. IEEE, 2011.

10

[25] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N Patt. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. In *ACM Sigplan Notices*, volume 45, pages 335–346. ACM, 2010.

10

[26] Stijn Eyerman and Lieven Eeckhout. Per-thread cycle accounting. *IEEE micro*, 30(1):71–80, 2010.

10

[27] Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 62–75. ACM, 2015.

11

[28] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 248–259. ACM, 2011.

11, 15, 24, 87

[29] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *ACM SIGARCH Computer Architecture News*, 41(1):77–88, 2013.

11, 24, 87

[30] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. In *ACM SIGPLAN Notices*, volume 49, pages 127–144. ACM, 2014.

11

[31] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 607–618, New York, NY, USA, 2013. ACM.

11, 27

[32] Jaeung Han, Seungheun Jeon, Young-ri Choi, and Jaehyuk Huh. Interference management for distributed parallel applications in consolidated

clusters. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 443–456. ACM, 2016.

12

[33] 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*, page 47. ACM, 2011.

12

[34] Lingjia Tang, Jason Mars, and Mary Lou Soffa. Contentiousness vs. sensitivity: improving contention aware runtime systems on multicore architectures. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, pages 12–21. ACM, 2011.

12, 62, 67

[35] Alexandra Fedorova, Sergey Blagodurov, and Sergey Zhuravlev. Managing contention for shared resources on multicore processors. *Communications of the ACM*, 53(2):49–57, 2010.

12, 63

[36] 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*, pages 379–391. ACM, 2013.

12, 86

[37] Joydeep Mukherjee, Diwakar Krishnamurthy, and Jerry Rolia. Resource contention detection in virtualized environments. *IEEE Transactions on Network and Service Management*, 12(2):217–231, 2015.

12

[38] Yasaman Amannejad, Diwakar Krishnamurthy, and Behrouz Far. Detecting performance interference in cloud-based web services. In *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 423–431. IEEE, 2015.

13

[39] Giuliano Casale, Carmelo Ragusa, and Panos Parpas. A feasibility study of host-level contention detection by guest virtual machines. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, volume 2, pages 152–157. IEEE, 2013.

13

[40] Jian Zhang and Renato J Figueiredo. Application classification through monitoring and learning of resource consumption patterns. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10–pp. IEEE, 2006.

15

[41] Qun Huang and Patrick PC Lee. An experimental study of cascading performance interference in a virtualized environment. *ACM SIGMETRICS Performance Evaluation Review*, 40(4):43–52, 2013.

17

[42] Wesam Dawoud, Ibrahim Takouna, and Christoph Meinel. Elastic vm for cloud resources provisioning optimization. In Ajith Abraham, Jaime Lloret Mauri, JohnF. Buford, Junichi Suzuki, and SabuM. Thampi, editors, *Advances in Computing and Communications*, volume 190 of *Communications in Computer and Information Science*, pages 431–445. Springer Berlin Heidelberg, 2011.

18, 50, 53

[43] Zhenhuan Gong, Xiaohui Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *Network and Service Management (CNSM), 2010 International Conference on*, pages 9–16, Oct 2010.

18, 50, 53, 82, 86

[44] 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 '11, pages 5:1–5:14, New York, NY, USA, 2011. ACM.

18, 50, 53, 82, 86

[45] Harold C Lim, Shivnath Babu, and Jeffrey S Chase. Automated control for elastic storage. In *Proceedings of the 7th international conference on Autonomic computing*, pages 1–10. ACM, 2010.

18, 50, 53, 71, 73, 74, 82, 86, 88, 102

[46] 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 10th International Conference on Autonomic Computing (ICAC 13)*, pages 69–82, San Jose, CA, 2013. USENIX.

18, 50, 53

[47] Scryer. `http://techblog.netflix.com/2013/11/scryer-netflixs-predictive-auto-scaling.html`.

18, 50, 53

[48] Ahmad Al-Shishtawy and Vladimir Vlassov. Elastman: autonomic elasticity manager for cloud-based key-value stores. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 115–116. ACM, 2013.

18, 50, 53, 71, 72, 74, 82, 86, 88

[49] N. Roy, A. Dubey, and A. Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 500–507, July 2011.

18, 50, 53

[50] Beth Trushkowsky, Peter Bodík, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. The scads director: Scaling a distributed storage system under stringent performance requirements. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

18, 50, 53, 71, 72, 73, 74, 88, 98

[51] Ying Liu, N. Rameshan, E. Monte, V. Vlassov, and L. Navarro. Prorenata: Proactive and reactive tuning to scale a distributed storage system. In

Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on, pages 453–464, May 2015.

18, 50, 53, 71, 72, 74, 88, 98

[52] Wikipedia Trace Data. `https://aws.amazon.com/datasets/6025882142118545`. 2012.

22

[53] Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Choosy: max-min fair sharing for datacenter jobs with constraints. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 365–378. ACM, 2013.

24

[54] Trevor F Cox and Michael AA Cox. *Multidimensional scaling*. CRC Press, 2010.

25

[55] David Williams and David Williams. *Weighing the odds: a course in probability and statistics*, volume 548. Springer, 2001.

25, 27

[56] Joshua B Tenenbaum, Vin De Silva, and John C Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, 2000.

25

[57] Timothy Sherwood, Erez Perelman, Greg Hamerly, Suleyman Sair, and Brad Calder. Discovering and exploiting program phases. *Micro, IEEE*, 23(6):84–93, 2003.

32

[58] LM Marsh and RE Jones. The form and consequences of random walk movement models. *Journal of Theoretical Biology*, 133(1):113–131, 1988.

33

[59] Ferdinando Urbano, Francesca Cagnacci, Clément Calenge, Holger Dettki, Alison Cameron, and Markus Neteler. Wildlife tracking data management: a new vision. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 365(1550):2177–2185, 2010.

33

[60] Edward Alexander Codling. *Biased random walks in biology.* PhD thesis, The University of Leeds, 2003.

34

[61] Michael F Shlesinger and Joseph Klafter. Lévy walks versus lévy flights. In *On growth and form*, pages 279–283. Springer, 1986.

34

[62] A Papoulis Probability. Random variables and stochastic processes. *McGrow, Hill Series Elastical Eng, NY*, 1984.

34

[63] Matt Williams and Tamara Munzner. Steerable, progressive multidimensional scaling. In *Information Visualization, 2004. INFOVIS 2004. IEEE Symposium on*, pages 57–64. IEEE, 2004.

36

[64] Tynia Yang, Jinze Liu, Leonard McMillan, and Wei Wang. A fast approximation to multidimensional scaling. In *Proceedings of the ECCV Workshop on Computation Intensive Methods for Computer Vision (CIMCV)*, pages 354–359, 2006.

36

[65] Linux Containers. 2012.

38

[66] Miguel G Xavier, Marcelo V Neves, Fabio D Rossi, Tiago C Ferreto, Timoteo Lange, and Cesar AF De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 233–240. IEEE, 2013.

38

[67] VLC. accessed:2014.

38

[68] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

38, 56, 90

[69] Cloud Suite Benchmark. accessed:2014.

38

[70] Jeanna Neefe Matthews, Wenjin Hu, Madhujith Hapuarachchi, Todd Deshane, Demetrios Dimatos, Gary Hamilton, Michael McCabe, and James Owens. Quantifying the performance isolation properties of virtualization systems. In *Proceedings of the 2007 workshop on Experimental computer science*, page 6. ACM, 2007.

38

[71] Confine Open Dataset. `https://wiki.confine-project.eu/experiments:datasets`. 2012.

39

[72] Amazon Elastic Compute Cloud. `http://aws.amazon.com/ec2/`.

50, 71, 81, 86

[73] Right Scale. `http://www.rightscale.com/`.

50, 71, 81, 86

[74] Simon J. Malkowski, Markus Hedwig, Jack Li, Calton Pu, and Dirk Neumann. Automated control for elastic n-tier workloads based on empirical modeling. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, pages 131–140, New York, NY, USA, 2011. ACM.

50, 82, 86

[75] Diego Didona, Paolo Romano, Sebastiano Peluso, and Francesco Quaglia. Transactional auto scaler: Elastic scaling of in-memory transactional data grids. In *Proceedings of the 9th International Conference on Autonomic Computing*, ICAC '12, pages 125–134, New York, NY, USA, 2012. ACM.

50, 82, 86

[76] X Zhu, D Young, BJ Watson, Z Wang, J Rolia, S Singhal, B McKee, C Hyser, D Gmach, R Gardner, et al. Integrated capacity and workload management for the next generation data center. In *ICAC'08: Proceedings of the 5th International Conference on Autonomic Computing*, 2008.

50, 82, 86

[77] Sujay Parekh, Neha Gandhi, Joseph Hellerstein, Dawn Tilbury, T Jayram, and Joe Bigus. Using control theory to achieve service level objectives in performance management. *Real-Time Systems*, 23(1-2):127–141, 2002.

50, 82, 86

[78] Khalid Alhamazani, Rajiv Ranjan, Karan Mitra, Fethi Rabhi, Prem Prakash Jayaraman, Samee Ullah Khan, Adnene Guabtni, and Vasudha Bhatnagar. An overview of the commercial cloud monitoring tools: research dimensions, design issues, and state-of-the-art. *Computing*, pages 1–21, 2014.

53

[79] Hamoun Ghanbari, Bradley Simmons, Marin Litoiu, and Gabriel Iszlai. Exploring alternative approaches to implement an elasticity policy. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 716–723. IEEE, 2011.

53

[80] Memcached. `http://memcached.org/`. accessed: April 2015.

56, 86

[81] Ravi Iyer, Ramesh Illikkal, Omesh Tickoo, Li Zhao, Padma Apparao, and Don Newell. Vm 3: Measuring, modeling and managing vm shared resources. *Computer Networks*, 53(17):2873–2887, 2009.

63

[82] Richard West, Puneet Zaroo, Carl A Waldspurger, and Xiao Zhang. Online cache modeling for commodity multicore processors. *ACM SIGOPS Operating Systems Review*, 44(4):19–29, 2010.

63

[83] Jason Mars, Lingjia Tang, and Mary Lou Soffa. Directly characterizing cross core interference through contention synthesis. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, pages 167–176. ACM, 2011.

63

[84] Pradeep Padala, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, and Kenneth Salem. Adaptive control of virtualized resources in utility computing environments. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 289–302, New York, NY, USA, 2007. ACM.

73

[85] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus. Using control theory to achieve service level objectives in performance management. *Real-Time Syst.*, 23(1/2):127–141, July 2002.

73

[86] Zhikui Wang, Xiaoyun Zhu, and Sharad Singhal. Utilization and slo-based control for dynamic sizing of resource partitions. In Jürgen Schönwälder and Joan Serrat, editors, *Ambient Networks*, volume 3775 of *Lecture Notes in Computer Science*, pages 133–144. Springer Berlin Heidelberg, 2005.

73

[87] Open Stack. `http://www.openstack.org`.

75

[88] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing*, 12(4):559–592, 2014.

81

[89] Abhishek Chandra, Weibo Gong, and Prashant Shenoy. Dynamic resource allocation for shared data centers using online measurements. In *Quality of Service—IWQoS 2003*, pages 381–398. Springer, 2003.

82

[90] Daniel Gmach, Jerry Rolia, Ludmila Cherkasova, and Alfons Kemper. Capacity management and demand prediction for next generation data centers. In *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pages 43–50. IEEE, 2007.

82

[91] Jiang Dejun, Guillaume Pierre, and Chi-Hung Chi. Ec2 performance analysis for resource provisioning of service-oriented applications. In *Service-Oriented Computing. ICSOC/ServiceWave 2009 Workshops*, pages 197–207. Springer, 2010.

86

[92] 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*, pages 301–312. ACM, 2014.

87

[93] MBW. `http://manpages.ubuntu.com/manpages/utopic/man1/mbw.1.html`. accessed: April 2015.

90

[94] Stream Benchmark. `http://www.cs.virginia.edu/stream/`. accessed: Feb 2015.

90

[95] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *nsdi*, volume 13, pages 385–398, 2013.

102