AUTONOMIC PERFORMANCE AND POWER CONTROL
IN VIRTUALIZED DATACENTERS

by

PALDEN LAMA

B.Tech., Indian Institute of Technology, India, 2003

A dissertation submitted to the Graduate Faculty of the
University of Colorado at Colorado Springs
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
Department of Computer Science
2013
Lama, Palden (Ph.D., Computer Science)

Autonomic Performance and Power Control in Virtualized Datacenters

Dissertation directed by Associate Professor, Chair Xiaobo Zhou

Virtualized datacenters, the platform for supporting Cloud computing, allow diverse applications to share the underlying server resources. Due to the highly dynamic nature of Internet workloads, increasing complexity of applications, and complex dynamics of shared infrastructure, datacenters face significant challenges in managing application performance while maintaining resource utilization efficiency and reducing power consumption costs. This thesis presents middleware approaches to autonomic performance and power control in virtualized datacenters. To this end, we designed self-adaptive resource management techniques based on queuing models, machine learning and feedback control theory.

Firstly, we designed an efficient server provisioning mechanism based on end-to-end resource allocation optimization for client perceived response time guarantee in a multi-tier server cluster. To guarantee an important percentile-based performance in the face of highly dynamic workloads, we developed a self-adaptive and model-independent neural fuzzy controller, which is capable of self-constructing and adapting its server allocation policies.

Secondly, we developed a non-invasive and energy-efficient mechanism for performance isolation of co-located applications on virtualized servers. Thirdly, we designed a system that provides coordinated power and performance control in a virtualized server cluster through a Fuzzy MIMO controller. We further developed a distributed and interference-aware control framework for large complex systems.

Furthermore, we developed a power-aware framework based on GPU virtualization for managing scientific workloads running in GPU clusters. It improves the system energy efficiency through dynamic consolidation and placement of GPU workloads.

Finally, we developed an automation tool for joint resource allocation and configuration of Hadoop MapReduce framework, for cost-efficient Big Data Processing in the Cloud. It addresses the significant challenge of provisioning ad-hoc jobs that have performance deadlines through a novel two-phase machine learning and optimization framework.

We implemented and evaluated the proposed techniques in a testbed of virtualized blade servers hosting
multi-tier applications, SPEC CPU2006 benchmarks, and Hadoop microbenchmarks. For evaluating power management of GPU clusters, we used NVIDIA Tesla C1060 GPUs. This thesis provides novel resource management solutions that control the quality of service provided by virtualized resources, improve the energy efficiency of the underlying system, and reduce the burden of complex system management from human operators.
Dedication

This thesis is dedicated to my parents for their constant support, encouragement and belief in my dreams.
Acknowledgements

This thesis would have been impossible without the support and mentoring of my advisor, Dr. Xiaobo Zhou. I would like to thank him for being a continuous source of encouragement, inspiration, and help for me to achieve my academic and career goals. I would also like to thank my graduate committee members, Dr. Boult, Dr. Chow, Dr. Yue, Dr. Rao and Dr. Zhang, for their help and encouragement throughout my Ph.D study at UCCS. Their valuable suggestions at the research proposal stage tremendously helped improve the quality of this thesis. I thank Dr. Pavan Balaji for giving me the opportunity to work with him as a research intern at Argonne National Laboratory. I also appreciate the camaraderie and help of my fellow DISCO lab members, Yanfei Guo, Dazhao Cheng, and Sireesha Muppala.

I am grateful to my parents for their encouragement and support in pursuing my dreams. I am very fortunate to have the continuous support of my wife, in the toughest of situations.

The research and dissertation were supported in part by the US National Science Foundation CAREER Award CNS-0844983 and research grants CNS-0720524 and CNS-1217979. I thank the College of Engineering and Applied Science for providing blade servers for conducting the experiments.
# Contents

1 Introduction 1

1.1 Performance and Power Management in Virtualized Datacenters 1

1.2 Motivation and Research Focus 3

1.2.1 Automated Scalability of Internet Services 3

1.2.2 Performance Isolation in Virtualized Datacenters 4

1.2.3 Coordinated Power and Performance Management 5

1.2.4 Power Management in High Performance Computing Datacenters 6

1.2.5 Big Data Processing in the Cloud 7

1.3 Challenges 7

1.3.1 Inter-tier Dependencies and Bottleneck Switch in Multi-tier Services 8

1.3.2 Complexity of Multi-service Applications 9

1.3.3 Non-linearity of Percentile-based Response Time 10

1.3.4 Highly Dynamic and Bursty workloads 11

1.3.5 The cost of Reconfiguration in datacenters 12

2 Related Work 14

2.1 Performance Management in Virtualized Platforms 15

2.1.1 Queuing Model Based Approaches 16

2.1.2 Control Theoretical Approaches 18

2.1.3 Machine Learning Based Approaches 20
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.1.4 Impact of the self-tuning controller on performance</td>
<td>61</td>
</tr>
<tr>
<td>3.4.1.5 Impact of optimization-fuzzy integration on resource allocation</td>
<td>62</td>
</tr>
<tr>
<td>3.4.2 Autonomic Performance Assurance with Self-Adaptive Neural Fuzzy Control</td>
<td>64</td>
</tr>
<tr>
<td>3.4.2.1 Effectiveness of Neural Fuzzy Control Approach</td>
<td>67</td>
</tr>
<tr>
<td>3.4.2.2 Comparison With Rule Based Fuzzy Controllers</td>
<td>70</td>
</tr>
<tr>
<td>3.4.2.3 Impact of Input Scaling Factor on Controller’s Self Adaptivity</td>
<td>73</td>
</tr>
<tr>
<td>3.4.2.4 Comparison with a PI controller under Varying Workload Characteristics</td>
<td>76</td>
</tr>
<tr>
<td>3.4.2.5 A Case Study based on the Testbed Implementation</td>
<td>78</td>
</tr>
<tr>
<td>3.5 Summary</td>
<td>83</td>
</tr>
<tr>
<td>4 Performance Isolation in Virtualized Datacenters</td>
<td>84</td>
</tr>
<tr>
<td>4.1 NINEPIN: Non-invasive and Energy efficient Performance Isolation</td>
<td>84</td>
</tr>
<tr>
<td>4.1.1 NINEPIN Architecture</td>
<td>86</td>
</tr>
<tr>
<td>4.1.1.1 Power and Performance Monitors</td>
<td>86</td>
</tr>
<tr>
<td>4.1.1.2 Level-1 Control</td>
<td>87</td>
</tr>
<tr>
<td>4.1.1.3 Level-2 Control</td>
<td>88</td>
</tr>
<tr>
<td>4.2 Performance Interference and Energy Usage Modeling</td>
<td>88</td>
</tr>
<tr>
<td>4.2.1 Fuzzy MIMO Model Formulation</td>
<td>89</td>
</tr>
<tr>
<td>4.2.2 Machine Learning Based Model Construction</td>
<td>90</td>
</tr>
<tr>
<td>4.2.3 Online Model Adaptation for Robust Performance Isolation</td>
<td>90</td>
</tr>
<tr>
<td>4.3 Utility Optimization</td>
<td>92</td>
</tr>
<tr>
<td>4.4 Model Predictive Controller</td>
<td>93</td>
</tr>
<tr>
<td>4.4.1 Linearized State-Space Model</td>
<td>93</td>
</tr>
<tr>
<td>4.4.2 MIMO Control Problem</td>
<td>95</td>
</tr>
<tr>
<td>4.4.3 Transformation to Quadratic Programming Problem</td>
<td>95</td>
</tr>
<tr>
<td>4.5 Implementation</td>
<td>96</td>
</tr>
<tr>
<td>4.5.1 The Testbed</td>
<td>96</td>
</tr>
<tr>
<td>4.5.2 NINEPIN Components</td>
<td>97</td>
</tr>
</tbody>
</table>
4.5.3 NINEPIN Overhead .............................................. 98

4.6 Evaluation ................................................................. 99

4.6.1 Performance Isolation .............................................. 99

4.6.2 Optimal Performance Targeting ................................. 101

4.6.3 System Utility and Energy Efficiency ......................... 102

4.6.4 NINEPIN Robustness .............................................. 103

4.7 Summary ................................................................. 105

5 Middleware for Power and Performance Control on Virtualized Servers .............................................. 106

5.1 PERFUME: Coordinated Power and Performance Control on Virtualized Server Clusters ... 107

5.1.1 PERFUME System architecture and Design ..................... 108

5.1.2 Modeling of Coordinated Power and Performance control .............. 109

5.1.2.1 The Fuzzy Model .............................................. 109

5.1.2.2 On-line Adaptation of the Fuzzy Model ......................... 112

5.1.2.3 Integration of workload-aware fuzzy modeling for proactive FUMI Control 113

5.1.3 FUMI Control Design ................................................ 115

5.1.3.1 FUMI Control Problem Formulation .......................... 115

5.1.3.2 Transformation to Quadratic Programming .................... 116

5.1.3.3 FUMI Control Interface ....................................... 120

5.1.4 Implementation ....................................................... 120

5.1.4.1 The Testbed .................................................... 120

5.1.4.2 PERFUME Components ..................................... 122

5.2 APPLEware: Autonomic Performance and Power Control for Co-located Web Applications on Virtualized Servers .............................................. 122

5.2.1 The Architecture ..................................................... 124

5.2.2 System Modeling ..................................................... 126

5.2.2.1 Global System Model ....................................... 126

5.2.2.2 Problem Decomposition ..................................... 127
5.2.3 Fuzzy Modeling To Capture System Non-linearity ........................................ 129
  5.2.3.1 Model Formulation ................................................................. 129
  5.2.3.2 Machine Learning Based Model Construction and Adaptation .............. 130
5.2.4 Distributed Control Design .............................................................. 130
  5.2.4.1 Control Formulation ............................................................... 131
  5.2.4.2 Distributed Control Algorithm .................................................. 131
  5.2.4.3 Speeding Up Local Control ....................................................... 131
  5.2.4.4 Computational Complexity Analysis ........................................... 134
5.2.5 Implementation ..................................................................................... 135
  5.2.5.1 Testbed ...................................................................................... 135
  5.2.5.2 APPLEware Components ......................................................... 136
5.3 Evaluation ............................................................................................... 137
  5.3.1 PERFUME ...................................................................................... 137
    5.3.1.1 Power and Performance Assurance with Flexible Tradeoffs ............ 137
    5.3.1.2 System Stability ................................................................. 139
    5.3.1.3 Percentile-Based Response Time Guarantee ................................ 140
    5.3.1.4 System Robustness under A Bursty Workload ............................ 140
    5.3.1.5 Impact of Proactive FUMI Control on Power and Performance Assurance 141
    5.3.1.6 Service Differentiation Provisioning ........................................ 142
    5.3.1.7 Effect of Control Parameter Tuning ........................................... 143
  5.3.2 APPLEware ....................................................................................... 145
    5.3.2.1 Model validation ................................................................. 145
    5.3.2.2 Autonomic Performance Control and Energy Efficiency .............. 146
    5.3.2.3 Scalability Analysis ............................................................... 149
5.4 Summary ................................................................................................. 150

6 Power-Aware Placement and Migration in High Performance Computing Datacenters 152
  6.1 GPU Virtualization .............................................................................. 153
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2</td>
<td>GPU Consolidation and Power-phase Awareness</td>
<td>154</td>
</tr>
<tr>
<td>6.3</td>
<td>pVOCL: Power-Aware Dynamic Placement and Migration in Virtualized GPU Environments</td>
<td>157</td>
</tr>
<tr>
<td>6.3.1</td>
<td>pVOCL Framework</td>
<td>157</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Power Modeling</td>
<td>158</td>
</tr>
<tr>
<td>6.3.2.1</td>
<td>Power-Phase Awareness</td>
<td>158</td>
</tr>
<tr>
<td>6.3.2.2</td>
<td>Analysis of Reconfiguration Overhead</td>
<td>159</td>
</tr>
<tr>
<td>6.3.3</td>
<td>pVOCL Components Design</td>
<td>160</td>
</tr>
<tr>
<td>6.3.3.1</td>
<td>Topology Monitor</td>
<td>160</td>
</tr>
<tr>
<td>6.3.3.2</td>
<td>Power Optimizer</td>
<td>161</td>
</tr>
<tr>
<td>6.3.3.3</td>
<td>Migration Manager</td>
<td>163</td>
</tr>
<tr>
<td>6.4</td>
<td>Evaluation</td>
<td>164</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Impact of GPU Consolidation</td>
<td>164</td>
</tr>
<tr>
<td>6.4.2</td>
<td>Power-Phase Topology Aware GPU Consolidation and Placement</td>
<td>166</td>
</tr>
<tr>
<td>6.4.3</td>
<td>Peak Power Management and Energy Efficiency</td>
<td>167</td>
</tr>
<tr>
<td>6.5</td>
<td>Summary</td>
<td>171</td>
</tr>
<tr>
<td>7</td>
<td>Automated Resource Allocation and Configuration of MapReduce Environment in the Cloud</td>
<td>172</td>
</tr>
<tr>
<td>7.1</td>
<td>Motivational Case Study</td>
<td>174</td>
</tr>
<tr>
<td>7.2</td>
<td>AROMA: Automated Resource Allocation and Configuration of MapReduce Environment in the Cloud</td>
<td>175</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Architecture and Design</td>
<td>175</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Machine Learning Performance Modeling</td>
<td>176</td>
</tr>
<tr>
<td>7.2.2.1</td>
<td>Data collection and clustering</td>
<td>176</td>
</tr>
<tr>
<td>7.2.2.2</td>
<td>Performance modeling</td>
<td>179</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Online Job Profiling and Signature Matching</td>
<td>181</td>
</tr>
<tr>
<td>7.2.4</td>
<td>Cost Efficient Performance Guarantee</td>
<td>182</td>
</tr>
<tr>
<td>7.3</td>
<td>Implementation</td>
<td>183</td>
</tr>
<tr>
<td>7.4</td>
<td>Evaluation</td>
<td>183</td>
</tr>
</tbody>
</table>
# List of Figures

1.1 Co-location of VMs on a multi-core processor. ........................................... 4
1.2 The CPU utilization of the front server and the database server across time with 1 second granularity [98, 97]. ................................................................. 8
1.3 Bottleneck shifting in independent tier specific server provisioning [123]. .............. 8
1.4 Multi-tier and Multi-service Architectures......................................................... 9
1.5 Multiple time-scale plots of the number of arriving HTTP requests [93]. The figure shows times scales of (a) one hour and (b) five seconds. ................................................. 11
1.6 Costs of a single VM migration on power consumption. ....................................... 12
1.7 Costs of a single VM migration on response time. ............................................. 12
2.1 A taxonomy of performance management techniques for Internet services ............. 14
3.1 A multi-tier server cluster architecture and end-to-end response time. ..................... 27
3.2 The flow chart and major steps of NSGA-II algorithm. ......................................... 34
3.3 Impact of the enhancement on the Pareto-optimal set. ......................................... 36
3.4 Impact of the enhancement on utilization of physical machines. ............................ 36
3.5 A model-independent fuzzy controller. .............................................................. 38
3.6 Fuzzy control effect. ......................................................................................... 39
3.7 The fuzzy rule base. ......................................................................................... 39
3.8 Membership functions for fuzzy control inputs $e(k)$ and $\Delta e(k)$. ....................... 40
3.9 Membership functions for control outputs. ......................................................... 41
3.27 Per-tier performance of neural fuzzy control for a dynamic workload. 68
3.28 End-to-end performance of neural fuzzy control for a stationary workload. 68
3.29 Per-tier performance of neural fuzzy control for a stationary workload. 69
3.30 $95_{th}$-Percentile end-to-end delay assurance for a dynamic step-change workload (target 1500 ms). 70
3.31 Performance comparison for various delay targets with dynamic step-change workload. 70
3.32 A continuously changing dynamic workload for a three-tier Internet service. 72
3.33 Median end-to-end delay for a continuously changing workload (target 1000 ms). 72
3.34 Performance comparison for various delay targets with continuously changing workload. 72
3.35 Performance comparison for various input scaling factors with delay target 1400 ms in case of dynamic step-change workload. 74
3.36 Performance comparison for various input scaling factors with delay target 1000 ms in case of dynamic step-change workload. 74
3.37 Performance comparison for various input scaling factors with delay target 1100 ms in case of continuously changing workload. 74
3.38 Performance comparison for various input scaling factors with delay target 1300 ms in case of continuously changing workload. 75
3.39 Performance comparison with PI control for varying workload characteristics. 77
3.40 End-to-end performance of neural fuzzy control based on a testbed implementation hosting RUBiS. 80
3.41 Per-tier performance of neural fuzzy control based on a testbed implementation hosting RUBiS. 81
3.42 End-to-end performance of neural fuzzy control with different control intervals. 82
4.1 NINEPIN: non-invasive performance isolation in virtualized servers. 85
4.2 NINEPIN System Architecture. 87
4.3 SPEC CPU2006 workload mixes. 97
4.4 Service-level utility of various SPEC CPU2006 applications. 98
4.5 Performance isolation by default, Q-Clouds and NINEPIN. 100

xvi
4.6 Optimal performance targets achieved by NINEPIN.

4.7 System utility and energy efficiency comparison between Q-Clouds and NINEPIN.

4.8 Prediction accuracy of NINEPIN’s performance and energy usage models.

4.9 NINEPIN robustness in the face of heterogeneous applications and dynamic workload variation.

5.1 PERFUME system overview.

5.2 The system architecture of PERFUME.

5.3 The prediction accuracy of performance and power by fuzzy models.

5.4 The prediction accuracy comparison between fuzzy and ARMA models under a dynamic workload.

5.5 Interface between FUMI control and learning components.

5.6 APPLEware: Autonomic Performance and Power Control.

5.7 APPLEware System Architecture.

5.8 APPLEware’s Average performance overhead.

5.9 Comparison of PERFUME and vPnP for control accuracy under a dynamic workload.

5.10 Comparison of PERFUME and vPnP for power and performance assurance under a dynamic workload.

5.11 The 95\textsubscript{th}-percentile response time.

5.12 Power and Performance assurance under a bursty workload generated by 1000 users.

5.13 Workload traffic trace and FUMI prediction.

5.14 Comparison of PERFUME with and without workload prediction of proactive FUMI control.

5.15 Service differentiation between two RUBiS applications for varying power budgets.

5.16 Performance results of PERFUME with different values of control penalty weight R.

5.17 FUMI control performance as a function of $H_p$.

5.18 APPLEware’s performance prediction (APP1) in the presence of interference effects.

5.19 Performance assurance under a stationary workload.

5.20 Performance and energy efficiency improvement due to APPLEware’s distributed control.

5.21 Robustness of APPLEware under dynamic and bursty workloads.
5.22 APLLEware performance assurance under dynamic and bursty workloads.

5.23 Performance and energy efficiency improvement due to APLLEware.

5.24 Testbed for scalability analysis.

5.25 APLLEware’ controller overhead with increasing number of applications.

6.1 The Virtual OpenCL (VOCL) framework.

6.2 Impact of phase imbalance on power efficiency. Efficiency is measured relative to the perfectly loaded configuration.

6.3 The original system configuration and two possible resulting configurations after VGPU migration and consolidation.

6.4 Impact of consolidation and node placement.

6.5 Power usage for various node configurations using two GPUs.

6.6 The Power-aware VOCL (pVOCL) framework.

6.7 Power profile of various node configurations.

6.8 Power profile of various application kernels using 2 GPUs and various node configurations.

6.9 Total execution time for each kernel over a range of input sizes with and without migration.

6.10 The topology monitor.

6.11 The migration manager.

6.12 Power consumption comparison with different power management techniques.

6.13 Energy usage and improvement for various workload mixes.

6.14 Impact of GPU consolidation on workload distribution across various power phases.

6.15 Impact of GPU consolidation on phase imbalance and node configurations.

6.16 Impact of power-phase topology aware consolidation and node placement on power consumption.

6.17 Power consumption trends under various peak power constraints.

6.18 Node configurations under various peak power constraints.

6.19 GPU workload distribution under various peak power constraints.

6.20 Performance overhead due to pVOCL under various power constraints.

xviii
6.21 Energy efficiency improvement due to pVOCL. ........................................... 170

7.1 The system architecture of AROMA. ................................................................. 175

7.2 CPU utilization of Sort, Wordcount and Grep jobs during different runs. .............. 177

7.3 Euclidean Signature Distances for CPU, Network and Disk resources for (i) Sort vs Sort;(ii) Sort vs Wordcount;(iii) Sort vs Grep; .................................................. 177

7.4 LCSS Signature Distances for CPU, Network and Disk resources for (i) Sort vs Sort;(ii) Sort vs Wordcount;(iii) Sort vs Grep; .................................................. 178

7.5 Prediction accuracy of AROMA performance model. ......................................... 181

7.6 Actual and predicted running times of MapReduce jobs for various VM resource allocations (small VMs). ................................................................. 184

7.7 Actual and predicted running times of MapReduce jobs for various VM resource allocations (medium VMs). ................................................................. 184

7.8 Actual and predicted running times of Sort for various input data sizes. ................ 184

7.9 Impact of AROMA’s auto-configuration on job performance and cost for various VM resource allocations (small VMs). .................................................. 185

7.10 Impact of AROMA’s auto-configuration on job performance and cost for various VM resource allocations (medium VMs). .................................................. 185

7.11 Impact of AROMA auto-configuration on job performance and cost for various input data sizes. 186

7.12 Prediction accuracy for an ad-hoc job. ............................................................ 189
# List of Tables

3.1 Notation Summary ................................................................. 50
3.2 The characteristics of workload A and B. ................................. 56
3.3 Workload characteristics A ..................................................... 65
3.4 Workload characteristics B ..................................................... 76

1 Performance of workload mix-1’s CPU2006 benchmark applications without performance isolation .................................................. 99
2 Performance targets for SPEC CPU2006 applications .................. 101
3 Utility and Energy Efficiency ...................................................... 102
4 Improvement in System Utility and Energy Efficiency .................. 104
1 APPLEware’s model validation for the multi-service application (App2). 145
1 The workload mix. ................................................................. 164
1 Cost and performance impact of resource allocation and parameter configuration. .... 175
2 Feature Selection for SORT performance model. ........................... 179
3 Feature Selection for GREP performance model. ........................... 179
4 Hadoop configuration parameters for Sort benchmark (six small VMs and 20 GB input data). 187
5 Hadoop configuration parameters for Sort benchmark (six medium VMs and 20 GB input data). 187
6 Cost of meeting job execution deadline (360 sec). .......................... 188
7 Cost of meeting job execution deadline (360 sec). .......................... 188

xx
Chapter 1

Introduction

1.1 Performance and Power Management in Virtualized Datacenters

Datacenters form the backbone of a wide variety of services offered via the Internet including Web-hosting, e-commerce, social networking, search engines, etc. They are found in nearly every sector of the economy: financial services, media, high-tech, universities, government institutions, and many others use and operate datacenters to aid business processes, information management, and communications functions. In the past decade, the number of datacenters operated by the U.S. government alone has skyrocketed from 432 to more than 1,200. Currently, there are 920 colocation datacenters from 48 states in USA, which provide shared infrastructure to multiple organizations [91].

Traditionally, datacenters are built on the over-provisioning model in which physical servers are allocated to handle the estimated peak demands of the hosted applications. Furthermore, separate servers are dedicated to host different applications as they run on different operating systems and also due to the need of performance isolation among critical applications. As a result, most servers in a typical datacenter run at only 5-10 percent utilization, offering a poor return on investment. At the same time, energy consumption costs and the impact of carbon footprint on the environment have become critical issues for datacenters today. In the United States alone, datacenters consumed $4.5 billion worth of electricity in 2006. A report by US Environmental Protection Agency (EPA) to the Congress reveals that the number of datacenter servers in the country increased from 4.9 million in 2000 to 10.5 million in 2006. Correspondingly, it estimated that the electricity
use of these servers increased from 11.6 billion kwh/year to 24.5 billion kwh/year during this period [36].

Today, datacenters are increasingly applying virtualization technology to achieve better server utilization and more flexible resource allocation for agile performance management. With virtualization, physical servers are provided as pools of logical computing capacity which can be divided into multiple virtual machines (VMs). These VMs can run multiple operating systems and applications as if they were running on physically separate machines. As a result, datacenters can consolidate a large number of physical machines into a small set of powerful servers, thereby improving server utilization and reducing power consumption costs. Virtualization is also a key enabling technology behind emerging cloud computing services such as infrastructure as a service (e.g., Amazon’s Elastic Compute Cloud (EC2) [1] and Simple Storage Service [2], Sun Grid [6], Rackspace [4]), software as a service (e.g., Microsoft Azure [7], Google App Engine [3]) and a number of others. Cloud computing services, which are built upon virtualized datacenters, allow customers to increase or decrease the amount of resources they want to reserve and pay for. It is made possible by the fact that VMs can grow or shrink in size and can be seamlessly moved from one physical server to another. Virtualized datacenters offer new opportunities as well as challenges in managing the performance of Internet services.

Computing systems have reached a level of complexity where the human effort required to get the systems up and running and keeping them operational is getting out of hand [51]. A large scale computing environment such as virtualized datacenters hosting multiple and heterogenous applications ranging from E-commerce to Big Data Processing, is a typical example of such complex system. To manage the performance of these applications manually demands extensive experience and expertise on the workload profile and on the computing system. However, the timescales over which the changes in the workload profile occur may not allow manual intervention. Furthermore, the contention of shared resources among multiple client applications that are consolidated on virtualized servers have a significant impact on the application performance. The situation is further complicated by the fact that datacenters need to control the power consumption to avoid power capacity overload, to lower electricity costs, and to reduce their carbon footprint. The complexity and the scale of virtualized datacenters make it increasingly difficult for administrators to manage them. Hence, there are growing research interests in autonomic computing paradigm in the context of modern datacenters.
Our main research goal is to develop autonomic performance and power control mechanisms based on efficient and self-adaptive resource management techniques. Towards this goal, we explore the use of queuing theoretical models, machine learning, feedback control techniques and hybrid approaches that integrate these techniques. We evaluate our proposed solutions through extensive simulations and implementation in our university prototype datacenter.

1.2 Motivation and Research Focus

We discuss the main motivations for autonomic performance and power control in virtualized datacenters and our research focus in detail.

1.2.1 Automated Scalability of Internet Services

Internet service providers often need to comply with quality of service requirements, specified in Service Level Agreement (SLA) contracts with the end-users, which determine the revenues and penalties on the basis of the achieved performance level. They want to maximize their revenues from SLAs, while minimizing the cost of resources used. Note that service providers may outsource their IT resource needs to public cloud providers, such as Google and Amazon or they may host their applications on their private virtualized datacenters. In order to meet performance SLAs, service hosting platforms often tend to over-provision the applications according to their peak expected resource demand. However, the resource demand of Internet applications can vary considerably over time.

Recent studies found highly dynamic and bursty workloads of Internet services that fluctuate over multiple time scales, which can have a significant impact on the processing demands imposed on datacenter servers [98, 97]. Modern datacenters are striving for automated scalability of Internet services that they host, in order to avoid resource under-utilization and inefficiency while providing performance assurance in the face of dynamic resource demands. An Internet service is scalable if it remains effective in performance when there is a significant increase of requests at the same time [144]. Automated scaling features are being included by some cloud vendors like Amazon and Right Scale [5]. However, they are based on static rules and policies specified by clients for individual VMs only. It is important and challenging to attain auto-
Figure 1.1: Co-location of VMs on a multi-core processor.

1.2.2 Performance Isolation in Virtualized Datacenters

Virtualization helps enable co-hosting of independent workloads by providing fault isolation, thereby preventing failures in one application from propagating to others. However, virtualization does not guarantee performance isolation between VMs [64]. It is mainly due to shared resource contention between VMs co-located in the same physical machine. For example, VMs residing on a multi-core processor share resources such as last level (LLC) cache, memory bandwidth, etc. to achieve better resource utilization and faster inter-core communication as shown in Figure 4.1. These VMs may experience significantly reduced performance when another VM simultaneously runs on an adjacent core, due to an increased miss rate in the last level cache (LLC) [38, 154]. A VM suffers extra cache misses because its co-runners (threads running on cores that share the same LLC) bring their own data into the LLC evicting the data of others. Contention for shared resources on multicore processors remains an unsolved problem in existing systems despite significant research efforts dedicated to this problem in the past [154]. Hence, it is challenging to achieving performance isolation between Internet applications running on virtualized datacenters.
Existing techniques such as resource partitioning of LLC cache aim to avoid performance interference among virtual machines. However, it leads to costly system complexity and inefficient resource utilization [100]. Some approaches consolidate applications according to their working set sizes for better performance isolation. However, a virtualized datacenter hosting third-party applications may not have such information. Furthermore, an application can manifest variable working sets at different stages of execution. Most of prior works focus on solutions that rely on either hardware level support or invasive instrumentation and modification of the guest operating system as well as virtualization management layer [38, 154]. A non-invasive solution for VM performance isolation can be more practical and easily deployable in virtualized datacenters.

1.2.3 Coordinated Power and Performance Management

Server virtualization has made significant contributions to the initiative towards Green datacenters. A key benefit of virtualization technology is the ability to contain and consolidate the number of servers in a datacenter. Ten server workloads running on a single physical server is typical, but some companies are consolidating as many as 30 or 40 workloads onto one server. Such dramatic reduction in server count has a transformational impact on IT energy consumption. Reducing the number of physical servers through virtualization cuts power and cooling costs and provides more computing power in less space. Many research studies and existing technologies focused on treating either power or performance as the primary control target in a datacenter while satisfying the other objective in a best-effort manner. Power oriented approaches [78, 101, 105, 129] disregard the SLAs of hosted applications while performance oriented approaches do not have explicit control on power consumption [20, 133]. Power consumption capping and performance assurance are inherently conflicting goals that can have various trade-offs. Hence, it is important to have a control mechanism that allows explicit coordination of power and performance in virtualized datacenters.

Today virtualized datacenters often consolidate workloads on high density blade servers, which impose stringent power and cooling requirements. It is essential to precisely control power consumption of blade servers to avoid system failures caused by power capacity overload or overheating. Furthermore, many datacenters are rapidly expanding the number of hosted servers while a capacity upgrade of their power dis-
tribution systems has lagged far behind. As a result, it can be anticipated that high-density server enclosures in future datacenters may often need to have their power consumption dynamically controlled under tight constraints [132]. However, existing power control techniques applied on server clusters may not be directly applicable to virtualized environments. Moreover, joint power and performance management solutions need to be accurate and stable even in the face of highly dynamic workload variation in virtualized datacenters.

1.2.4 Power Management in High Performance Computing Datacenters

General-purpose graphics processing units (GPUs) have rapidly gained popularity as accelerators for core computational kernels across a broad range of scientific, engineering, and enterprise computing applications. They are ubiquitous accelerators in high performance computing datacenters today [10, 40, 109, 112, 119]. It is mainly due to their excellent performance-to-power ratio, which comes from a fundamental restructuring of the processor hardware layout, consisting of thousands of efficient cores designed for parallel performance. The advent of general-purpose programming models, such as CUDA and OpenCL, has further accelerated the adoption of GPUs by simplifying the parallelization of many applications and high-level libraries on them.

While GPUs can deliver much higher performance than CPUs, it comes at the cost of significantly higher power consumption. The thermal design power (TDP) of a high-end GPU, e.g. 512-core NVIDIA Fermi, is as large as 295 watts(W), while a high-end quad-core x86-64 CPU has a TDP of 125 watts. Hence, the usage of power-hungry GPUs in the already power-consuming high performance computing systems must be carefully evaluated with respect to the impacts on overall system power efficiency.

There are significant challenges in achieving online power management of GPU-enabled server clusters in a datacenter environment. Today, most datacenter cabinets are equipped with 3-Phase Cabinet Power Distribution Units (CDUs) to cater for increased power demands, greater equipment densities and cost reduction initiatives. However, as an artifact, the underlying system infrastructure shows complex power consumption characteristics depending on the placement of GPU workloads across various compute nodes, power-phases and cabinets. In addition, the power drawn across the three phases in the same cabinet needs to be balanced for better power efficiency and equipment reliability. Furthermore, power delivery and cooling limitations in datacenters impose peak power constraints at various levels. For instance, server racks are typically pro-
visioned for 60 Amps of current. This could become a bottleneck for high density configurations, specially when power-hungry GPUs are used.

### 1.2.5 Big Data Processing in the Cloud

Today, there is a deluge of data, growing at an exponential rate in various sectors of the economy. This data comes from everywhere: sensors used to gather climate information, posts to social media sites, digital pictures and videos, purchase transaction records, and cell phone GPS signals to name a few. Large-scale distributed data processing in enterprises is increasingly facilitated by software frameworks such as Google MapReduce and its open-source implementation Hadoop, which parallelize and distribute jobs across large clusters [9, 28, 146]. There are growing interests in deploying such a framework in the Cloud to harness the unlimited availability of virtualized resources and pay-per-usage cost model of cloud computing. For example, Amazon’s Elastic MapReduce provides data processing services by using Hadoop MapReduce framework on top of their compute cloud EC2, and their storage cloud S3.

Existing MapReduce environments for running Hadoop jobs in a cloud platform aim to remove the burden of hardware and software setup from end users. However, they expect end users to determine the number and type of resource sets to be allocated and also provide appropriate Hadoop parameters for running a job. A resource set is a set of virtualized resources rented as a single unit, e.g., virtual machines rented by Amazon web services. Here, we use the term resource set and virtual machine interchangeably. In the absence of automation tools, currently end users are forced to make job provisioning decisions manually using best practices. As a result, customers may suffer from a lack of performance guarantee and increased cost of leasing the cloud resources.

### 1.3 Challenges

We discuss several challenges encountered in achieving autonomic performance and power control in virtualized datacenters.
1.3.1 Inter-tier Dependencies and Bottleneck Switch in Multi-tier Services

Popular Internet applications hosted in a datacenter have complex multi-tier architecture. In a multi-tier Internet application, each tier may impose different resource demands depending on the workload intensity and characteristics. As a result, the resource bottleneck may lie at different tiers. For example, the CPU of database server is usually the bottleneck for the online bookstore benchmark, TPC-W [114] whereas the auction site benchmark [110], RUBiS saturates the server at the frontend. Furthermore, recent studies found that multi-tier services exhibit a phenomenon called bottleneck switch in which resource saturation occurs alternately at different tiers across time [98, 97]. It is due to the presence of burstiness in the service times of requests processed at various tiers. Figure 1.2 shows the bottleneck switch effect in terms of varying CPU utilizations of the front server and the database server for a browsing workload mix of TPC-W benchmark.

Figure 1.2: The CPU utilization of the front server and the database server across time with 1 second granularity [98, 97].

(a) Bottleneck at Tier 2. (b) Bottleneck shifts to Tier 3.

Figure 1.3: Bottleneck shifting in independent tier specific server provisioning [123].
Although single-tier server provisioning mechanisms are well-studied, its straightforward extension to performance management of multi-tier services is not effective. A recent study demonstrated the independent server provisioning at the bottleneck tier does not necessarily improve the performance of the multi-tier application [123]. Instead, it merely shifts the bottleneck to the downstream tier. For example, consider a three-tier Internet application depicted in Figure 1.3 (a). Initially, assume that one server each is allocated to the three tiers, and this enables the application to service 15, 10 and 10.5 requests/sec at each tier. Let the incoming request rate be 14 requests/sec. Given the above capacities, all requests are let in through the first tier, and 4 requests/sec are dropped at the second tier. Due to these drops, the third tier sees a reduced request rate of 10 requests/sec and is able to service them all. Thus, the effective throughput is 10 requests/sec. Since request drops are only seen at the second tier, this tier is perceived to be the bottleneck. The provisioning algorithm at that tier will allocate an additional server, doubling its effective capacity to 20 requests/sec. At this point, the first two tiers are able to service all incoming requests and the third tier now sees a request rate of 14 requests/sec (see Figure 1.3 (b)). Since its capacity is only 10.5 requests/sec, it drops 3.5 requests/sec. Thus, the bottleneck shifts to the third tier, and the effective throughput only increases from 10 to 10.5 requests/sec.

1.3.2 Complexity of Multi-service Applications

Traditional multi-tier web applications have a simple pipelined architecture in which each tier provides certain functionality to its preceding tier and uses the functionality provided by its successor to carry out its part of the overall request processing. This is illustrated by Figure 1.4(a). Today, large enterprise applications are increasingly incorporating a service-oriented architecture (SOA) style, in which modular components are composed to implement the business logic. Major web sites such as Amazon, eBay, etc have moved from monolithic 2-tier/3-tier architecture to a multi-service architecture for better scalability and manageability.
Such an architecture comprises of a complex set of disparate and collaborating services which are usually stateful and have interdependencies.

Figure 1.4(b) shows an example of a multi-service application. The root service invokes the left branch for gathering user information, then the right branch for promoting product information to the same user. The User info service in turn accesses the shared data service, then invokes an external XSLT service to transform XML templates into HTML. The Promotion service in the right branch first fetches users order histories from the shared data service, then searches for items related to users last orders using the Product data service in order to recommend further purchases. Finally, the root service combines the results from the two branches in one web page and returns it to the client.

Resource provisioning for effective performance management of multi-service applications is challenging due to its complex inter-service relationships [55]. The situation is further complicated by the fact that a virtualized datacenter often co-host multi-tier as well as multi-service applications.

### 1.3.3 Non-linearity of Percentile-based Response Time

It is very challenging to assure a percentile-based response time guarantee of requests of a multi-tier Internet service. Compared with the average response time, a percentile response time introduces much stronger nonlinearity to the system, making it difficult to derive an accurate performance model. A nonlinear system is a system which does not satisfy the superposition principle, or whose output is not directly proportional to its input. The variable(s) to be solved in a such system cannot be written as a linear combination of independent components. In general, non-linear equations that define a system are difficult to solve. Hence, it is difficult to estimate the resource capacity needs of Internet services that is required to assure percentile based performance guarantee. Queueing theoretic techniques have achieved noteworthy success in providing average delay guarantee on multi-tier server systems. However, queueing models are mean oriented and have no control on percentile-based delay. Recently, control theoretic techniques were applied to inherently nonlinear Web systems for performance guarantees by performing linear approximation of system dynamics and estimation of system parameters [60]. However, if the deployed system configuration or workload range deviates significantly from those used for system identification, the estimated system model used for control
Figure 1.5: Multiple time-scale plots of the number of arriving HTTP requests [93]. The figure shows times scales of (a) one hour and (b) five seconds.

would become inaccurate [89]. Traditional control theoretic techniques may not be effective in achieving percentile-based response time guarantee for Internet services in virtualized datacenters, which impose highly variable resource demands.

1.3.4 Highly Dynamic and Bursty workloads

Internet workloads show highly dynamic variation in its intensity as well as characteristics. The workload intensity which is usually measured in terms of request arrival rate vary at multiple time scales. Figure 1.5 demonstrates the workload variability of e-commerce applications through a workload characterization study conducted on an online bookstore [93]. The data comprises two weeks of accesses to each of these sites. The bookstore logs were collected from August 1st to August 15th, 1999, while the auction server logs are from March, 28th to April 11th, 2000. During these two weeks, the bookstore handled 3,630,964 requests (242,064 daily requests on average), transferring a total of 13,711 megabytes of data (914 MB/day on average). Another important phenomenon called burstiness or temporal surges in the incoming requests in an e-commerce server generally turns out to be catastrophic for performance, leading to dramatic server overloading, uncontrolled increase of response times and, in the worst case, service unavailability. Traffic surges may be caused by unforeseeable events such as stock markets roller coaster ride, terror attacks, Mars landing, etc. or by Slashdot effect, where a web page linked by a popular blog or media site suddenly experiences a huge increase of the number of hits. Auction sites (e.g., eBay) where users compete to buy an object that is going to be soon assigned to the customer with the best offer, and e-business sites with special offers and marketing campaigns
may also face bursty workloads. In such environments, autonomic performance control mechanism needs to be robust and self-adaptive to dynamic variations in workload.

1.3.5 The cost of Reconfiguration in datacenters

In virtualized datacenters, dynamic resource provisioning can be performed through various adaptation actions such as: increase/decrease a VMs CPU capacity by a fixed amount, addition/removal of a VM, live-migration of a VM between hosts, and shutting down/restarting physical hosts. Addition of a VM replica is implemented by migrating a dormant VM from a pool of VMs to the target host and activating it by allocating CPU capacity. A replica is removed by migrating it back to the pool.

Dynamic resource reconfiguration actions come with associated costs. Server switching by addition and removal of a virtual server introduces non-negligible latency to a multi-tier service, which will affect the perceived end-to-end response time of users. For example, an addition of database replica goes through a data migration and system stabilization phase. A removal of a server does not happen instantaneously, since it has to process residual requests of an active session. Furthermore, server provisioning during an adaptation phase will cause oscillations in performance [19].

Although state of the art virtualization technology has reduced the downtime during VM migration to a few hundred milliseconds [23], the end-to-end performance and power consumption impacts can still be
significant. A recent study [56] measured the increase in power consumption and end-to-end response time of a 3-tier Web/Java/MySQL application as a function of time during the live migration of a single of its Xen-based VMs. As shown in Figures 1.6 and 1.7 the measurements taken for three different workloads of 100, 400, and 800 concurrent user sessions, illustrates the significance of VM migration impact as well as its dependence on workload.
Chapter 2

Related Work

Recent research efforts have explored mechanisms for managing the performance of Internet services, with a goal to provide QoS guarantees to customers. The proposed mechanisms have been evolving according to the progress of Internet services as well as the platform that host them. For instance, as Internet services transformed from single tier to multi-tier architectures, new schemes have been proposed to tackle the ensuing challenges. Similarly, the advent of virtualized platforms for hosting Internet applications gave rise to novel techniques that utilize the opportunities provided by virtualization while dealing with related challenges. As shown in Figure 2.1, the techniques can be classified according the platform where they are applied such as traditional server systems and virtualized platforms; theoretical foundations that drive the decision making process such as queuing models, feedback control and machine learning; various performance goals including

![Figure 2.1: A taxonomy of performance management techniques for Internet services](image-url)
mean based and percentile based throughput and response time guarantee. In this section, we discuss some of the performance management techniques that are closely related to our work.

Power management of server systems has received a lot of research interests due to three significant reasons. First, there are power delivery and cooling limitations in a datacenter environment due to increasing power density of high performance servers, which potentially leads to power capacity overload or overheating [101, 106]. Second, electricity costs associated with energy consumption and the cost of cooling infrastructure constitute a significant portion of datacenter operating cost. [27, 105]. Finally, there is a motivation for building Green datacenters to protect the environment. We discuss some recent works on power management of datacenter servers.

2.1 Performance Management in Virtualized Platforms

Virtual machine (VM) technology is widely adopted as an enabler of cloud computing. In a typical virtualized environment, a hypervisor executes on a physical machine and presents an abstraction of the underlying hardware to multiple virtual machines (VMs). The hypervisors support lifecycle management functions for the hosted VM images, and facilitation of both offline and live migration of the execution environment for the VM [12, 127]. Virtualization provides many benefits, including scalability, improved resource utilization, ease of management and flexibility in resource allocation. Allocating new resources in virtualized environment becomes much faster, as on-the-fly cloning of hundreds of virtual machine can happen within sub-second [68]. Furthermore, state of the art virtualization technology has reduced the downtime during VM migration to a few hundred milliseconds [23].

Recently, significant research has been conducted in performance management of Internet services in virtualized platforms. Menascé and Bennani considered dynamic priority scheduling and allocation of CPU shares to virtual servers [94]. Wang et al. proposed a virtual-appliance-based autonomic resource provisioning framework for large virtualized datacenters [130]. Weng et al. designed a management framework for a virtualized cluster system, and presented an automatic performance tuning strategy to balance the workload [139]. The work in [103] presented a resource control system, AutoControl, which can detect and mitigate CPU and disk I/O bottlenecks that occur over time and across multiple nodes in shared virtualized
infrastructure by allocating each resource accordingly. Watson et al. modeled the probability distributions of performance metrics, in terms of percentiles, based on variables that can be readily measured and controlled in a virtualized environment [135]. The work in [140] designed an automated approach for profiling different types of virtualization overhead on a given platform and a regression-based model that maps the native system profile into a virtualized one.

A few studies focused on static and dynamic server consolidation techniques based on virtualization [14, 57, 96, 115, 124, 141]. Static consolidation technique utilizes the historical information of average resource utilizations for mapping VMs to appropriate physical machines. After initial static consolidation the mapping may not be recomputed for long periods of time, such as several months, and is done off-line. Sonnek and Chandra [115] identified VMs that are most suitable for being consolidated on a single host. They propose to multiplex VMs based on their CPU and I/O boundedness, and to co-locate VMs with higher potential of memory sharing. Meng et al. [96] exploited statistical multiplexing of VMs to enable joint VM provisioning and consolidation based on aggregated capacity needs.

In contrast, dynamic consolidation operates on shorter timescales and leverages the ability to do live migration of VMs [14]. For instance, the Sandpiper system proposed in the work [141] automates the detection of hotspots and determines if VMs should be migrated by monitoring their memory utilizations. Jung et al. [57] tackled the problem of optimizing resource allocation in consolidated server environments by proposing a runtime adaptation engine that automatically reconfigures multi-tier web applications running in virtualized datacenters while taking into account adaptation costs and thus satisfying response-time-based SLAs even under rapidly changing dynamic workloads.

This thesis focuses on application-centric resource management techniques on virtualized platforms, which aim at satisfying application level performance while reducing the costs of resource reconfiguration and achieving performance isolation between co-located VMs in the presence of shared resource contention.

2.1.1 Queuing Model Based Approaches

A large number of works propose queuing model-based approaches to achieve the performance guarantees in Internet systems. Their basic idea is to model the system behavior using a queuing model and use the
classical results from queuing theory to predict the resource management actions necessary to achieve the specified performance targets given the current observed workload. Earlier works applied queueing models for resource allocation optimization of single-tier Internet servers [126, 151, 153]. For example, the work in [126] studied an optimization for allocating servers in the application tier that increase a server provider's profits. An optimization problem is constructed in the context of a set of application servers modeled as $M/G/1$ processor sharing queueing systems. That single-tier provisioning method does not consider the end-to-end response time constraint.

Recently, there are a few studies on the modeling and analysis of multi-tier servers with queueing foundations [13, 30, 31, 34, 86, 87, 113, 116, 123, 122]. Stewart and Shen [117] proposed a profile-driven performance model for cluster-based Internet services. Application profiling was done offline. Liu et al. [86] proposed an analytical model of a three-tier Web service. The mean-value analysis algorithm for queueing networks was used to measure the average end-to-end delay. Diao et al. [31] described a performance model based on $M/M/1$ queueing for differentiated services of multi-tier applications. Per-tier concurrency limits and cross-tier interactions were addressed in the model. Villela et al. [126] studied optimal server allocation in the application tier that increase a server provider's profits. An optimization problem is constructed in the context of a set of application servers modeled as $M/G/1$ processor sharing queueing systems. The work in [122] proposed an analytic model for session-based multi-tier applications using a network of queues. The mean-value analysis algorithm for queueing networks was used to measure the mean response time. Singh et al. [113] proposed a novel dynamic provisioning technique that handles both the non-stationarity in the workload and changes in request volumes when allocating server capacity in datacenters. It is based the k-means clustering algorithm and a $G/G/1$ queuing model to predict the server capacity for a given workload mix.

Urgaonkar et al. designed an important dynamic provisioning technique on virtualized multi-tier server clusters [123]. It sets the per-tier average response time targets to be certain percentages of an end-to-end response time bound. Based on a queueing model, per-tier server provisioning is executed at once for the per-tier response time guarantees. The work provides important insights on dynamic virtual server provisioning for multi-tier clusters. There is however no guidance nor optimization about the decomposition of end-to-end response time to per-tier response time targets. This thesis proposes an efficient server provisioning approach
on multi-tier clusters based on an end-to-end resource allocation optimization model.

Although queuing model based resource provisioning approach [16, 30, 55, 31, 103, 117, 126] is effective in controlling the average response time of Web requests under steady-state conditions, it does not easily extend to managing the actual distribution of response time. Today, many Internet applications require a strong guarantee on the tail of the response time distribution. We design a model independent fuzzy controller to guarantee the 95th-percentile response time and integrate it with end-to-end optimization model for resource allocation efficiency.

2.1.2 Control Theoretical Approaches

Feedback control has been used in real-time systems for long time. A typical feedback controller controls the parameters of the performance management action using feedback information from the system, while providing guarantee on system stability and responsiveness. Lu et al. designed an utilization control algorithm (EUCON) for distributed real time systems in which each task is comprised of a chain of subtasks distributed on multiple processors [90]. It is based on a model predictive control approach that models utilization control on a distributed platform as a multi-variable constrained optimization problem. Wang et al. extended it to a decentralized algorithm, called DEUCON [131]. In contrast to the centralized control schemes, DEUCON features a novel decentralized control structure that requires only localized coordination among neighbor processors.

Recent research efforts have proposed the use of control theory for performance management in the context of Internet applications [8, 60, 89]. Linear control techniques were applied to control the resource allocation in single-tier Web servers [8]. However, the performance of the linear feedback control is often limited [136]. Karma et al. [60] designed a proportional integral (PI) controller based admission control proxy to bound the average end-to-end delay in a three-tier Web service. There are studies that argue model-dependent control techniques may suffer from the inaccuracy of modeling dynamic workloads in multi-tier systems. For instance, Lu et al. [89] modeled a controlled Web server with a second order difference equation whose parameters were identified using the least square estimator. The estimation was performed for a certain range and characteristics of workload. The estimated system model used for control would become inaccurate
if the real workload range deviates significantly from those used for performance model estimation [89]. We propose a model-independent fuzzy control for server allocation in multi-tier clusters, which is free from the ill-effects of modeling inaccuracies.

Fuzzy theory and control were applied for Web performance guarantee due to its appealing feature of model independence, and used to model uncertain and imprecise information in applications [150]. Liu et al. [88] used fuzzy control to determine an optimal number of concurrent child processes to improve the Apache web server performance. Wei and Xu [136] designed a fuzzy controller for provisioning guarantee of user-perceived response time of a web page. Those fuzzy controllers were designed manually on trial and error basis. Important design parameters such as input scaling factors, rule base and membership functions are not adaptive. They are not very effective in the face of highly dynamic workloads. This thesis presents the design of a self-adaptive neural fuzzy controller which is robust to highly dynamic workloads.

Recently, multiple-input-multiple-output (MIMO) control technique has been applied for performance management of Internet applications [67, 129, 133]. A key advantage of having a control foundation is its theoretically guaranteed control accuracy and system stability. In addition, MIMO based approaches can handle the complexity of multi-tier service architecture such as inter-tier dependency, bottleneck switching, as well as the system dynamics of virtualized environments. However, these approaches are designed based on offline system identification for specific workloads [67, 129, 133]. Hence, they are not adaptive to situations with abrupt workload changes though they can achieve control accuracy and system stability within a range theoretically. Padala et al. [103] proposed AutoControl, a combination of an online model estimator and a multi-input multi-output controller. The resource allocation system can automatically adapt to workload changes in a shared virtualized infrastructure to achieve the average response time based service level objective. However, using the average response time as the performance metric is unable to represent the shape of a response time curve [138]. We design a Fuzzy MIMO controller, which can guarantee the percentile-based response time of multi-tier applications while dynamically adapting a fuzzy performance model of the system in response to dynamic workload variations.
2.1.3 Machine Learning Based Approaches

Machine learning techniques have drawn significant research interests for autonomic performance management of Internet services. Given the complexity of Internet services and the underlying infrastructure that host them, machine learning based performance management approach is attractive as it assumes little or no domain knowledge and it can adapt to changes in the system and its environment. Recently, machine learning techniques have been used for measuring the capacity of Internet websites [88, 108], for online hardware reconfiguration [15, 107] and for autonomic resource allocation [121, 147].

Bu et al. [15] proposed a reinforcement learning approach for autonomic configuration and reconfiguration of multi-tier web systems. In [121], a hybrid of queuing models and reinforcement learning was proposed for autonomic resource allocation. In their approach, reinforcement learning initially trains offline on data collected while a queuing model policy controls the system. It aims to avoid potentially poor performance in live online training. Similar reinforcement learning strategy is also used for virtual machine auto-configuration by VCONF [107]. It automates the VM configuration and dynamically reallocates the resources allocated to VMs in response to the change of service demands or resources supply.

Work by Cohen et al. [24] used a probabilistic modeling approach called Tree-Augmented Bayesian Networks (TANs) to identify combinations of system-level metrics and threshold values that correlate with high-level performance states - compliance with service-level agreements for average response time - in a three-tier Web service under a variety of conditions. Experiments based on real applications and workloads indicate that this model is a suitable candidate for use in offline fault diagnosis and online performance prediction. One approach [108] applied a bayesian network to correlate low level instrumentation data such as system and user CPU time, available memory size, and I/O status that are collected at run-time to high level system states in each tier of a multi-tier web site. A decision tree was induced over a group of coordinated bayesian models in different tiers to identify the bottleneck dynamically when the system is overloaded.

Singh et. al [113] proposed an autonomic mix-aware dynamic provisioning technique that applied the k-means clustering algorithm to automatically determine the workload mix of non-stationary workloads. The work in [19] applied the K-nearest-neighbors (KNN) machine learning approach for allocating database replicas in dynamic content Web server clusters. Experiments using the TPC-W e-commerce benchmark
demonstrated the benefits of proactive resource provisioning approach based on the KNN technique.

This thesis presents self-adaptive and robust performance control mechanisms by integrating machine learning and control theoretical techniques. Machine learning promises self-adaptiveness in the face of dynamic workloads and control theoretical foundation promises system stability and control accuracy.

### 2.1.4 Percentile Based Performance Guarantee

Percentile-based performance metric has the benefit that is both easy to reason about and to capture individual users’ perception of Internet service performance [69, 80, 123, 135, 138]. Welsh and Culler [138] proposed to bound the 90\textsuperscript{th}-percentile response time of requests in a multi-stage Internet server. It is achieved by an adaptive admission control mechanism that controls the rate of request admission. The mechanism complements, but does not apply to dynamic server provisioning in datacenters.

Urgaonkar et al. [123] proposed an interesting approach for assuring the 95\textsuperscript{th}-percentile delay guarantee. It uses an application profiling technique to determine a service time distribution whose 95\textsuperscript{th}-percentile is the delay bound. The mean of that distribution is used as the average delay bound. It then applies the bound for the per-tier delay target decomposition and per-tier server provisioning based on a queueing model. There are two key problems. One is that the approach is queueing model dependent. The second is that the application profiling needs to be done offline for each workload before the server replication and allocation. Due to the very dynamic nature of Internet workloads, application profiling itself can be time consuming and importantly not adaptive online.

Leite et al. [80] applied an innovative stochastic approximation technique to estimate the tardiness quantile of response time distribution, and coupled it with a proportional-integral-derivative (PID) feedback controller to obtain the CPU frequency for single-tier servers that will maintain performance within a specified deadline. It is non-trivial to apply this approach to dynamic server allocation problem. First, it does not compensate for the effect of process delay in resource allocation, which is significant due to server switching costs. The controller was designed solely based on response time measurement and manual tuning of controller parameters for a particular simulated workload. As a result, it may not be adaptive to highly dynamic workloads.
Watson et al [135] proposed an unique approach to model the probability distributions of response time, in terms of percentiles, based on CPU allocations on virtual machines. The performance model was obtained by offline training based on data collected from the system. It is not adaptive online to dynamically changing workloads. The work focuses on performance modeling without addressing issues related to adaptive resource provisioning such as process delay, system stability, performance assurance, etc.

We apply fuzzy theory and control for dealing with the non-linearity of percentile based performance metric such as the 95th- percentile response time. Fuzzy rules are able to represent various regions of the complex non-linear system model using a simple functional relations.

2.2 Power Management

Power management in computing devices and systems is an important and challenging research area. There were many studies in power management in standalone, battery-operated, embedded mobile devices. For instance, the Dynamic Voltage Scaling (DVS) technique was integrated with real-time schedulers to provide energy savings while maintaining hard and soft deadline guarantees of embedded systems [145], applied to reduce power consumption in Web servers [35], and utilized to improve power efficiency of server farms [39].

Today, popular Internet applications have a multi-tier architecture forming server pipelines. Applying independent DVS algorithms in a pipeline will lead to inefficient usage of power for assuring an end-to-end delay guarantee due to the inter-tier dependency [50]. Wang et al. [129] proposed a MIMO controller to regulate the total power consumption of an enclosure by conducting processor frequency scaling for each server while optimizing multi-tier application performance. Such controllers are designed based on offline system identification for specific workloads. They are not adaptive to situations with abrupt workload changes though they can achieve control accuracy and system stability within a range theoretically.

Modern datacenters apply virtualization technology to consolidate workloads on fewer powerful servers for improving server utilization, performance isolation and flexible resource management. Traditional power management techniques are not easily applicable to virtualized environments where physical processors are shared by multiple virtual machines. For instance, changing the power state of a processor by DVS will inadvertently affect the performance of multiple virtual machines belonging to different applications [101,
133].

It is a trend that power and performance management of virtualized multi-tier servers are jointly tackled. However, it is challenging due to the inherently conflicting objectives.

*Power-oriented approaches* aim to ensure that a server system does not violate a given power budget while maximizing the performance of hosted applications [79, 78, 102, 101, 105, 132, 129] or increasing the number of services that can be deployed [37, 44]. pMapper [124] tackles power-cost tradeoffs under a fixed performance constraint. vManage [66] performs VM placement to save power without degrading performance. Co-Con [132] is a novel two-level control architecture for power and performance coordination in virtualized server clusters. It gives a higher priority to power budget tracking and performance is a secondary goal.

*Performance-oriented approaches* aim to guarantee a performance target while minimizing the power consumption [20, 54, 67, 75, 80, 99, 133]. However, they do not have explicit control over power consumption.

*Coordinated power and performance management with explicit trade-offs* is recently studied in virtualized servers [18, 43, 56, 61, 62]. The work in [61] proposed an approach for semantics-free coordination between power and performance modules. Mistral [56] is a control architecture to optimize power consumption, performance benefit, and the transient costs incurred by adaptations in virtualized server clusters. vPnP [43] coordinates power and performance in virtualized servers using utility function optimization. It provides the flexibility to choose various tradeoffs between power and performance. However, it lacks the guarantee on system stability and performance, especially under highly dynamic workloads. The trade-off flexibility and system stability requirements in the face of highly dynamic workloads, together with the percentile-based response time guarantee, demand novel techniques for autonomous performance and power control.

We develop a system, PERFUME, which controls both power and performance with flexible trade-offs while assuring system stability. It handles both average and percentile based performance targets.


2.3 Performance Isolation in Virtualized Datacenters

Performance isolation of customer applications in a virtualized datacenter is an important research topic. Despite several advantages such as security isolation, fault isolation, and environment isolation, prevalent virtualization techniques do not provide effective performance isolation between VMs [64, 100]. The behavior of one VM can affect the performance of another adversely due to the shared use of resources in the system. VMs running on the underlying multi-core servers of a virtualized datacenter mainly suffer from the performance interference caused by the contention of last level cache and memory bandwidth. The performance impact of shared resource contention in multi-core servers has been well studied in the studies [46, 64, 100].

Several research efforts have focused on hardware and software resource partitioning based techniques for performance isolation of applications running on a multi-core server. Hardware-based cache partitioning schemes are mainly involved with modification of cache replacement policies [149] with various partition granularity such as cache ways and cache blocks. On the other hand, software partitioning technique based on static and dynamic page coloring addresses cache contention between competing applications, without requiring any hardware level support [22, 118, 148]. Page coloring reserves a portion of the cache for each application, and allocates the physical memory such that the application’s cache lines map only into the reserved portion. However, such approaches in virtualized servers require invasive instrumentation and modification of the guest operating system or the virtualization management layer.

Some prior studies investigated the design of cache-aware scheduling algorithms that achieves performance isolation among competing applications by minimizing resource contention [38, 63, 154]. For instance, Fedorova et. al designed a cache-aware scheduler that compensates threads that were hurt by cache contention by giving them extra CPU time [38]. Knauerhase et. al [63] proposed to reduce cache interference by spreading the cache intensive applications apart and co-scheduling them with non-intensive applications. A common drawback of cache-aware scheduling and resource partitioning based performance isolation mechanism is that they only focus on a single source of performance interference. However, in practice there are several dimensions of performance interference such as shared I/O and memory bandwidths [64].

Recently, Nathuji et. al. proposed an interesting non-invasive performance isolation approach for virtualized servers, Q-Clouds [100]. Q-Clouds builds MIMO models that capture interference relationships between co-
located VMs and applies a closed loop controller to achieve specified performance levels for each VM. Due to its non-invasive nature, the approach does not need to determine the underlying sources of interference. However, it disregards the economic objective of a datacenter, which is defined by the service-level utility of customer applications. Furthermore, it does not consider energy efficiency and heterogeneous application support.

Energy consumption costs and the impact of carbon footprint on the environment have become critical issues for datacenters today [39, 81]. There are recent studies that aim to guarantee fixed performance targets of datacenter applications while minimizing the power consumption [20, 54, 67, 72, 75, 80]. However, they do not consider the impact of performance interference between co-located VMs on the energy efficiency and the system utility.

2.4 Big Data Processing in the Cloud

Recently distributed data processing framework MapReduce and its open source implementation Hadoop have gained much research [41, 49, 76, 77, 104, 111]. The pay-per-use utility model of Cloud computing introduces new opportunities as well as challenges in deploying a MapReduce framework [134]. Lee et al. designed an architecture to allocate resources to a data analytics cluster in the cloud, and proposed a metric of share in a heterogeneous cluster to realize a scheduling scheme that achieves high performance and fairness [76]. However, their approach allocates resources based on a simple criteria of job storage requirement without considering any performance guarantee.

There are a few studies focusing on performance estimation and guarantee of MapReduce jobs. Polo et al. [104] proposed an online job completion time estimator that can be used for adjusting the resource allocations of different jobs. However, their job estimator tracks the progress of the map stage alone and has no information or control over the reduce stage. An interesting approach recently designed by Verma et al. uses job profiles of routinely executed Hadoop jobs and a MapReduce performance model to determine the amount of resources required for job completion within a given deadline [125]. However, such an approach does not guarantee the performance of ad-hoc jobs submitted to the system. More importantly, it does not consider the impact of Hadoop configuration parameters on the effectiveness and efficiency of resource allocation.
Resource allocation efficiency in datacenters is very important from the economical perspective [42, 143].

Kambatla et al. [59] proposed to select the optimal Hadoop configuration parameters for improving the performance of jobs using a given set of resources. However, there is no guidance on deciding the appropriate number and type of resources to be allocated. There is a need for a holistic system that considers the interdependence of various job provisioning decisions in providing performance guarantee in a cost efficient manner.

There are simulation based approaches to systematically understanding the performance of MapReduce setups [45] and tuning the MapReduce configuration parameters [48]. However, designing an accurate simulator that can comprehensively capture the internal dynamics of such complex systems is potentially error-prone.

There are recent studies that address the important issue of improving MapReduce query performance [25, 32]. For example, Lee et al. proposed and developed YSmart, a correlation aware SQL-to-MapReduce query translator [77]. Those studies are complimentary to our research.

We develop an automation tool for joint resource allocation and configuration of MapReduce environment in the Cloud. It is able to provision ad-hoc MapReduce jobs for achieving performance guarantee in a cost-efficient manner.
Chapter 3

Autonomic Computing in Virtualized Environments

Autonomic computing aims at enabling modern, networked computing systems to manage themselves without direct human intervention. It is inspired by the autonomic nervous system of the human body. This nervous system controls important bodily functions (e.g. respiration, heart rate, and blood pressure) without any conscious intervention. In a datacenter, such self-managing capability is essential to reduce the burden of complex system management from human operators and administrators. To this end, we propose and develop efficient and self-adaptive resource provisioning techniques for performance assurance of various applications hosted in a virtualized datacenter, in the face of dynamic execution environments.

3.1 Resource Allocation Optimization with Performance Guarantee

End-to-end response time is the major performance metric of multi-tier Internet applications. It is the response time of a request that flows through a multi-tier system. Figure 3.1 illustrates an example of a typical three-tier server cluster. However, as resource usage comes with associated costs, an efficient resource allocation mechanism that satisfies performance guarantee is very desirable in a virtualized datacenter. Recently,

![Figure 3.1: A multi-tier server cluster architecture and end-to-end response time.](image)

...
an important dynamic virtual server provisioning approach was proposed in [123] for multi-tier Internet applications. The approach decomposes an end-to-end response time bound into the per-tier average response time targets (i.e., \( d_1 \), \( d_2 \), and \( d_3 \)). Then per-tier server provisioning is conducted based on a queueing model to meet the per-tier response time target. Its key problem, however, is on how to determine those decomposition percentages, while the dynamic behavior of an Internet application shifts the performance bottleneck from one tier to another. The important performance metric is the end-to-end response time, not the per-tier response time. The thesis here is that in the absence of any specific guideline, the target decomposition based server provisioning may lead to inefficient resource usage.

We propose an efficient server provisioning technique based on queueing theoretical optimization model for end-to-end response time guarantee on multi-tier clusters in a virtualized environment. The optimization model attempts to minimize the total number of virtual servers allocated to a multi-tier cluster while the end-to-end response time guarantee is satisfied. The basic idea is to divide the provisioning process into a sequence of intervals. In each interval, based on the measured resource utilization, end-to-end response time, and the predicted workload, the servers are allocated to the tiers at once. The workload of each virtual server at each tier can be modeled by one M/G/1 queueing system. We consider the use of homogeneous virtual servers. First, as others in [123], we assume that requests at a virtual server are processed with the FCFS principle. Then, we consider the principle of processor sharing for concurrent request processing at a virtual server. We explore both FCFS and processor sharing disciplines with the optimization model.

### 3.1.1 End-to-end Optimization with Queueing Modeling

Popular multi-tier Internet applications are session based [95, 123, 153]. Consider sessions arrive at a rate \( \lambda \) to a \( n \)-tier server cluster. Let \( v_i \) denote the average number of visits of a session to tier \( i \). The request arrival rate to tier \( i \) becomes \( \lambda v_i \). Note that the average number of visits per session fluctuates substantially over time for many applications. A request in different tiers usually demands different processing resource usages [95, 153, 123]. Let \( r_i \) be the normalized resource demand of a request in tier \( i \). Let \( m_i \) be the number of servers allocated to tier \( i \) in the current interval. With a load balancer, the workload at a tier is shared by the allocated homogeneous servers. Let \( \rho_i \) be the resource utilization of tier \( i \), \( \rho_i = \lambda v_i r_i / m_i \). The requests will
traverse multiple tiers. A recent study based on actual traces of request arrivals to the application tier of an e-commerce site shows that the arrival process is effectively Poisson [126]. Like many works in [30, 126, 153], we assume that request arrivals to a tier meet a Poisson process.

We first model the workload of each virtual server at each tier by one $M/G/1$ FCFS queueing system. Let $X_i$ be the service time distribution at tier $i$. Let $d_i$ denote the average response time of a request in tier $i$. According to Pollaczek-Khinchin formula of the queueing theory, we have

$$d_i = E[W_i] + E[X_i] = \frac{\rho_i E[X_i^2]}{2r_i(1 - \rho_i)} + E[X_i],$$

where $E[W_i]$ is the expected queueing delay at tier $i$, $E[X_i]$ and $E[X_i^2]$ are the first moment and second moment of the service time distribution $X_i$ at tier $i$, respectively. The formula follows from the fact that $W_i$ and $X_i$ are independent from a FCFS queue [47].

We formulate the dynamic server provisioning with the end-to-end response time guarantee as the following optimization problem:

$$\text{Minimize} \quad \sum_{i=1}^{n} m_i \quad (3.1)$$

Subject to

$$\sum_{i=1}^{n} d_i = U - \bar{U} \quad (3.2)$$

$$d_i = \frac{\rho_i E[X_i^2]}{2r_i(1 - \rho_i)} + E[X_i] \quad (3.3)$$

$$m_i = \frac{\lambda v_i r_i}{\rho_i} \quad (3.4)$$

$$0 \leq \rho_i < 1. \quad (3.5)$$

Eq. (3.1) gives the optimization objective. It is to minimize the total number of servers allocated to the multi-tier server cluster. We assume the servers are homogeneous with the same cost and each tier is replicable on-demand [19]. If one wants to differentiate the deployment costs at different tiers, the objective function can be a weighted sum. Eq. (7.3) describes that the average end-to-end response time of a request is bounded by $U$. $\bar{U}$ is an important control parameter. If it equals to zero, the average end-to-end response time of a request is no more than the bound $U$. If it equals to $\epsilon$, the average end-to-end response time of a request is less than the bound $U$. We later use it for the integration of the optimization with a model-independent fuzzy controller for resource allocation efficiency. Eq. (3.3) describes the predicted average response time of a request at a tier. Note that $\rho_i = \lambda v_i E[X_i]$. Eq. (3.4) describes the relationship between the server allocation
and the utilization of each tier. It assumes a load balancer which can equally distribute the workload to each
virtual server of a tier. Eq. (3.5) describes the resource allocation constraint.

The equations (3.1) and (3.4) lead to the objective function

\[ L(\rho_1, \cdots, \rho_n) = \sum_{i=1}^{n} \frac{\lambda_{i} r_{i}}{\rho_{i}}. \]  

(3.6)

The equations (7.3) and (3.3) lead to the constraint function

\[ C(\rho_1, \cdots, \rho_n) = \sum_{i=1}^{n} \left( \frac{\rho_{i} E[X_{i}^2]}{2r_{i}(1-\rho_{i})} + E[X_i] \right). \]  

(3.7)

Let \( \nu \) be the Lagrange multiplier. The optimal solution to (3.1) occurs when the first order derivatives of
the Lagrangian objective function \( L(\rho_1, \cdots, \rho_n) \) over variables \( \rho_i \) are equivalent to the first order derivatives
of the constraint function \( C(\rho_1, \cdots, \rho_n) \) over variable \( \rho_i \) multiplied by \( \nu \). That is

\[ \frac{\partial L(\rho_1, \cdots, \rho_n)}{\partial \rho_i} = \nu \frac{\partial C(\rho_1, \cdots, \rho_n)}{\partial \rho_i}. \]  

(3.8)

Deriving \( \nu \) and using it with (3.4) leads to the solution

\[ \rho_i = \left( 1 + \frac{\sum_{i=1}^{n} \sqrt{\lambda_{i} E[X_{i}^2]} - \sum_{i=1}^{n} E[X_i]}{2} \right)^{-1} \]  

(3.9)

and

\[ m_i = \lambda_{i} r_{i} + \frac{\sum_{i=1}^{n} \sqrt{\lambda_{i} E[X_{i}^2]} - \sqrt{\sum_{i=1}^{n} E[X_i]} \sqrt{\lambda_{i} E[X_{i}^2]}}{2}. \]  

(3.10)

Internet workload measurements indicate that a heavy-tailed distribution is often an accurate model for
service time distribution for many Web applications. Like work in [47, 137], we use a bounded pareto
distribution for modeling a heavy-tailed distribution. Our previous work reported in [137] found closed-form
expressions of \( E[X] \) and \( E[X_{i}^2] \) for a bounded pareto distribution, which are used in this optimization model.

We next consider the principle of processor sharing for concurrent request processing at a virtual server.
Processor sharing principle accounts for multi-threaded processors concurrently handling requests at a virtual
server. Let \( X_i \) be the service time distribution at tier \( i \). Let \( d_i \) denote the average response time of a request
in tier \( i \). Based on the queueing foundations, we have

\[ d_i = \frac{E[X_i]}{(1-\rho_i)}. \]
where \( E[X_i] \) is the expected first moment of the service time distribution \( X_i \) at tier \( i \).

Applying the Lagrange multiplier method, we come up with the solution for the processor sharing based optimization. That is

\[
\rho_i = \left( 1 + \frac{\sum_{i=1}^{n} E[X_i] \sqrt{\lambda v_i}}{U - \sum_{i=1}^{n} E[X_i] \sqrt{\lambda v_i}} \right)^{-1} 
\]

\[
m_i = \lambda v_i r_i + \frac{\sum_{i=1}^{n} E[X_i] \sqrt{\lambda v_i}}{U - \sum_{i=1}^{n} E[X_i] \sqrt{\lambda v_i}} E[X_i] \sqrt{\lambda v_i}.
\]

### 3.1.2 Multi-objective Server Provisioning

Server provisioning in a modern datacenter built upon virtualized server clusters for hosting Internet applications, is a highly complex task as it involves satisfying multiple correlated and conflicting objectives in a dynamic and often unpredictable environment. For example, resource allocation efficiency and performance assurance are two important but essentially conflicting objectives. As it is increasingly difficult for human operators to manage such complex systems manually, there are significant research interests in developing autonomic solutions for performance control in modern datacenters [13, 26, 27, 65, 94, 103, 113, 116, 120, 128, 130].

Autonomic resource management for achieving multiple objectives of a datacenter is often enabled by utility computing paradigm. Utility functions represents various degrees of desirability for different QoS levels. However, it is difficult to define a local utility function for one objective of an application and determine the weights on each utility to form a global utility function. We argue that it may even be inefficient to choose static weights, given the fact that Internet applications face dynamic workload variations.

We address the issue of autonomic server provisioning in a virtualized multi-tier cluster environment from a novel perspective. Unlike a utility-based approach, we treat each objective as a separate entity in the optimization without applying any pre-determined weights. We formulate a multi-objective optimization problem using analytic queueing network model. This can be solved using a non-dominated sorting based genetic algorithm, which will result in a set of Pareto-optimal solutions that offer various tradeoffs between multiple objectives.
3.1.2.1 Multi-objective optimization problem

A modern datacenter built upon virtualized server clusters for hosting multi-tier applications has multiple objectives. Each application competes for shared resources for QoS provisioning to its users. From the perspective of the datacenter, virtual servers need to be allocated efficiently. Under-utilized resources in physical machines become a liability issue to the datacenter because of inefficient power consumption, space utilization, and excessive cost of ownership. We thus consider three important objectives:

1. Minimize the total number of physical machines used for all applications.
2. Minimize the average system end-to-end response time, a key QoS metric, for all applications.
3. Minimize the total number of virtual servers allocated to all applications to free up more physical machines and also to reduce virtualization overhead.

There are constraints due to limited resources and QoS needs. We consider five important constraints as follows:

1. The average end-to-end response time of each application must be below a given bound according to the service level agreement [69, 123].
2. The total number of virtual servers running for all applications on one physical machine must not exceed a specified limit due to the concurrency limit [30].
3. The total number of virtual servers allocated for one application and running on one physical machine must not exceed a bound to ensure performance isolation.
4. The utilization of a virtual server cannot exceed its resource capacity limit.
5. The number of physical machines available is limited.

We consider a virtualized datacenter that has $N$ physical machines virtualized and shared by $M$ applications with $K$ tiers. Let $a_{ijk}$ be the number of virtual servers allocated to tier $k$ of application $j$ and placed in physical machine $i$. Let $\rho_{jk}$ be the resource utilization of application $j$ at tier $k$ in a virtual server, and $d_{jk}$ be the average response time of requests served at tier $k$ of application $j$. The average end-to-end response time experienced by requests flowing through multiple tiers is the sum of average response time at
each tier [69, 70, 123]. Let $U_j$ be the average end-to-end response time target of an application $j$ and $W$ be the virtual machine allocation limit of a physical server. We formulate dynamic server provisioning as a multi-objective optimization problem as follows:

Minimize $m$  

Minimize $\sum_{j=1}^{M} \sum_{k=1}^{K} d_{jk}$  

Minimize $\sum_{i=1}^{N} \sum_{j=1}^{M} \sum_{k=1}^{K} a_{ijk}$  

Subject to Constraints:

$\forall j \in [1, M], \sum_{k=1}^{K} d_{jk} \leq U_j$  

$\forall i \in [1, N], 0 \leq \sum_{j=1}^{M} \sum_{k=1}^{K} a_{ijk} \leq W$  

$\forall j \in [1, M], \forall k \in [1, K], 0 \leq \rho_{jk} < 1$  

$m \leq N.$

Eqs. (3.13), (3.14) and (3.15) give the optimization objectives. Eqs. (3.17), (3.18), (3.19) and (3.20) define the four constraints. The number of physical machines used is given by $m = \sum_{i=1}^{N} f(a_{ijk})$ where

$$
 f(a_{ijk}) = \begin{cases} 
 1 & \text{if } (\sum_{j=1}^{M} \sum_{k=1}^{K} a_{ijk}) \geq 1 \\
 0 & \text{if } (\sum_{j=1}^{M} \sum_{k=1}^{K} a_{ijk}) < 1 
\end{cases}
$$

3.1.2.2 Obtaining a Pareto-Optimal set

Classical optimization methods suggest converting a multi-objective optimization problem to a single-objective problem which aims to optimize the weighted sum of multiple objectives. Such methods assume that the weights to be assigned are well known in advance. However, in practice, it is difficult to choose proper weights that will result in an efficient solution to the optimization problem. More important and challenging is that choosing weights statically may not be efficient in the face of dynamic workloads. We want to achieve an automated server provisioning scheme that provides a solution with the most efficient trade-off between multiple objectives. As the first step towards achieving this goal, we solve the multi-objective server provisioning problem by applying a non-dominated sorting based optimization algorithm. This results in a Pareto-optimal set of solutions, which offer various tradeoffs between performance and server allocation.
Begin
Create initial population of size N
Perform fast non-dominated sorting of population P
Calculate crowding distance
Perform selection using crowded-comparison operator
Perform crossover and mutation to obtain population Q
Combine populations P and Q
Pick the best N solutions from the combined population
If Gen > Max Gen
Yes
No
End

Figure 3.2: The flow chart and major steps of NSGA-II algorithm.
We apply a computationally fast multi-objective genetic algorithm (NSGA-II) [29] to obtain multiple Pareto-optimal solutions in one single run. Note that there are other algorithms applicable to our multi-objective optimization problem. In order to apply the genetic algorithm, we represent a solution to the optimization problem by a “chromosome”. It is a string of numbers, coding information about the decision variables. The decision variables in our multi-objective optimization problem are the elements of a 3-dimensional matrix of size $N \times M \times K$. An element is denoted as $a_{ijk}$. For encoding the decision variables in a chromosome, we convert the 3-dimensional matrix to a chromosome vector of length $N \times M \times K$. It is denoted as $A = (a_1, a_2, ..., a_{N \times M \times K})$. As a result, the element $a_{ijk}$ of the original decision matrix is equal to the $(i \times M \times K + j \times K + k)^{th}$ element of vector $A$. It is known as a gene in the chromosome.

Figure 3.2 shows the major steps of the NSGA-II algorithm.

### 3.1.2.3 Enhancements on Multi-Objective Optimization

We further enhance the multi-objective optimization algorithm used for obtaining a Pareto-optimal set of solutions, utilizing the system knowledge about practical server switching cost and behavior of a virtualized multi-tier system. Note that the enhancement techniques are applicable to any heuristic search technique such as simulated annealing, genetic algorithm, etc.

**Improving usage of physical machines** One main advantage of server virtualization for resource allocation is the improvement in the utilization efficiency of physical resources [130, 139]. We emphasize the advantage by applying a threshold-based enhancement on the optimization algorithm. The enhancement feeds incremental search space to the genetic algorithm for finding the Pareto-optimal set. It initiates the genetic algorithm for multi-objective optimization using a small fraction of the total number of physical machines available. The physical machines are chosen randomly for this purpose, assuming that available resources are homogenous due to virtualization. The algorithm searches for the Pareto-optimal set within this confined search space. After a certain number of evaluations, it increases the number of physical machines in the search space by one if the percentage of evaluations that violated the resource constraint defined in Eq. (3.18) exceeds a certain threshold.

Based on simulation, we consider a cluster of 20 physical machines shared by an application with an
end-to-end response time target of 300 ms. Let the total number of virtual servers allocable to one application on a physical machine be up to 60. For a particular workload, we compare the Pareto-optimal set obtained by the optimization with the enhancement and without the enhancement. Figure 3.3 shows that compared to the optimization without the enhancement, the Pareto-optimal set obtained by the optimization with the enhancement uses less than half of the number of physical machines for achieving the same range of average end-to-end response time $[247.89 \text{ ms} \sim 299.04 \text{ ms}]$.

Figure 3.4 illustrates that the solutions obtained with the enhancement have higher average utilization of physical machines. Experimental results indicate that the developed enhancement improves utilization of physical machines, which in turn frees up more of the available physical machine pool.

**Reducing server switching cost** Server switching by addition and removal of a virtual server at a tier introduces non-negligible latency to a multi-tier service. It affects the perceived average response time of users. A newly added server spends time adapting to the existing system. For example, an addition of
database replica goes through a data migration and system stabilization phase [19], during which the delay will be higher than expected. A removal of a server does not happen instantaneously, since it has to process residual requests of an active session. We enhance the optimization algorithm by reducing the undesired effects of server-switching cost. The enhancement incorporates the time required for the reconfiguration of server allocation scheme into calculation of the average response time, while evaluating a group of candidate solutions. A candidate solution provides a potential server configuration scheme. Consider that the addition of a virtual server at tier $k$ to an application needs time $T_{ak}^a$ and the removal of virtual server needs time $T_{rk}^r$.

Given $N$ physical machines and $M$ applications, the server switching cost in terms of delay is calculated by:

$$T_c = \max_{i \in [1,N]} \left( \sum_{j=1}^{M} \sum_{k=1}^{K} ((b_{ijk} - a_{ijk})^* \cdot T_{ak}^a + (a_{ijk} - b_{ijk})^* \cdot T_{rk}^r) \right),$$

where $a_{ijk}$ is the current virtual server configuration and $b_{ijk}$ is a candidate solution. $(b_{ijk} - a_{ijk})^*$ represents $\max(0, b_{ijk} - a_{ijk})$. For application $j$, let $D_j^a$ and $D_j^b$ be the average end-to-end response time of the current virtual server configuration and the candidate solution respectively. Let $T_s$ be the “control interval” at which reconfiguration decision is made periodically. Considering the time required for reconfiguration, the enhancement calculates the expected average end-to-end response time of the application as follows:

$$D_j = \frac{T_c \cdot D_j^a + (T_s - T_c) \cdot D_j^b}{T_s}, \forall j \in [1, M].$$

This enhancement essentially penalizes too frequent server switching. It plays an important role in avoiding any potential system thrashing that could result from dynamic workload fluctuations.

### 3.2 Model-independent fuzzy control for Percentile-Delay Guarantee

Traditional resource provisioning approaches for Internet applications focus on providing average performance guarantees for Web requests. However, they are not very effective in controlling the tail of the response time distribution. It is mainly due to the fact that percentile-delay metrics have a complex and highly non-linear relationship with resource allocation. We propose a model-independent fuzzy controller to bound the $90_{th}$-percentile response time of requests on a multi-tier server architecture. Unlike classical control theoretical techniques, a fuzzy control based server provisioning technique does not require any explicit performance model of the system. Hence, it is free from the ill effects of modeling inaccuracies that could arise.
Figure 3.5: A model-independent fuzzy controller.

due to the complexity of multi-tier web systems and workload variations. The fuzzy controller uses a set of heuristic rules that determine the number of servers to be allocated at each tier to bound the 90th-percentile end-to-end response time. It has a self-tuning component based on a scaling-factor controller, designed to compensate for the cost of reconfiguration due to the addition or removal of a server at a tier.

Figure 3.5 illustrates the architecture of the model-independent fuzzy controller. The controller has two inputs; $e(k)$ is the difference between the target value and measured value of the end-to-end response time in the $(k)th$ sampling period (target response time - measured response time), and $\Delta e(k)$ is the change in error. The output of the controller is the resource adjustment $\Delta m(k)$ for the next sampling period. The scaling factors $K_e$, $K_{\Delta e}$ and $\alpha K_{\Delta m}$ are used to tune the controller’s performance. $\alpha$ is the output scaling factor. Thus, the total number of servers allocated to the multi-tier clusters during the $(k+1)th$ sampling period is

$$m(k+1) = m(k) + \alpha K_{\Delta m} \Delta m(k) = \int \alpha K_{\Delta m} \Delta m(k) dk. \quad (3.21)$$

The number of servers allocated to a specific tier is determined by the ratio of average response time of that tier to the sum of the average response time of all tiers. This allocation of servers based on the response time proportion is intuitive because the tier with the highest response time may be facing the highest workload. Hence, more servers should be allocated to that tier.

The fuzzy controller consists of four components. The rule base is the core component. It contains a set of rules based on which fuzzy control decisions are made. The fuzzification interface converts numeric values of controller inputs into equivalent fuzzy values. It determines the certainties of fuzzy values based on input membership functions. The inference component applies pre-defined rules according to the fuzzified inputs and generates fuzzy conclusions. The defuzzification interface combines fuzzy conclusions and converts them to a single output, i.e., the resource allocation adjustment in a numeric value.
3.2.1 The Fuzzy Rule Base

Designing the rule base for a fuzzy controller is based on heuristic control knowledge. Hence, it requires a number of experiments to come up with a good set of rules with trials and errors. The rules are defined using linguistic variables “$e(k)$”, “$\Delta e(k)$” and “$\Delta m(k)$” corresponding to the numeric values of control inputs and outputs. The linguistic variables “$e(k)$” and “$\Delta e(k)$” have linguistic values NL, NM, NS, ZE, PS, PM, and PL, which stand for negative large, negative medium, negative small, zero, positive small, positive medium and positive large respectively. The linguistic variable “$\Delta m(k)$” has two additional linguistic values denoted by NH and PH, which stand for negative huge and positive huge respectively. We choose a larger set of linguistic values for the fuzzy controller output in order to increase the flexibility in the adjustment of server allocation. The rules are in the form of If-Then statements. For example, If error “$e(k)$” is NL and change in error “$\Delta e(k)$” is PL, then the server allocation adjustment “$\Delta m(k)$” is ZE.

To design the fuzzy control rules, we analyze the behavior of end-to-end response time due to changes
in resource allocation. We identify five zones of the end-to-end response time as shown in Figure 3.6. Since $e(k)$ and $\Delta e(k)$ have opposite signs in zones 1 and 3, the error is self-correcting. If $e(k)$ is small, $\Delta m(k)$ needs to be adjusted to slow down the current trend so as to avoid any overshoot. For example, if “$e(k)$” is $PS$ and “$\Delta e(k)$” is $NL$, then “$\Delta m(k)$” is $PL$. Whereas, if $e(k)$ is large, $\Delta m(k)$ needs to be adjusted to speed up the current trend. For example, if “$e(k)$” is $PL$ and “$\Delta e(k)$” is $NS$, then “$\Delta m(k)$” is $NL$. In zones 2 and 4, $e(k)$ and $\Delta e(k)$ have the same sign. That is, the measured end-to-end response time is moving away from the target value. Therefore, $\Delta m(k)$ should be adjusted to reverse the current trend. For example, if “$e(k)$” is $PL$ and “$\Delta e(k)$” is $PL$, it means that the measured response time is smaller than the target and the trend is continuing. This scenario is depicted by zone 2. Thus, “$\Delta m(k)$” is $NH$ so as to decrease the number of servers allocated and bring the actual response time closer to the target. Zone 5 indicates the steady state since $e(k)$ and $\Delta e(k)$ have small magnitudes. In this case, $\Delta m(k)$ should be adjusted to maintain the current state and to correct steady state errors. For example, if “$e(k)$” is $PS$ and “$\Delta e(k)$” is $NS$, then “$\Delta m(k)$” is $ZE$.

The control rules designed for each of the analyzed zones are illustrated in Figure 3.7.

3.2.2 Fuzzification, Inference, Defuzzification

Fuzzification is the process of converting the numeric input values into corresponding linguistic values and calculating the certainty of those linguistic values. The effective inputs to fuzzy controller are $e(k)$ multiplied by input scaling factor $K_e$ and $\Delta e(k)$ multiplied by input scaling factor $K_\Delta e$. The linguistic values are represented by membership functions. Membership function is a graphical representation of the magnitude of participation of each input. As work in [136], we choose triangle membership functions due to the simplicity and wide usage. We consider both uniform and non-uniform membership functions. Like in others
Figure 3.9: Membership functions for control outputs.

work [137], the membership functions for both scaled inputs and output are defined within the common interval [-1,1]. The values of the inputs $e(k)$ and $\triangle e(k)$ are mapped into [-1,1] by the input scaling factors $K_e$ and $K_{\triangle e}$, respectively. The output value $\triangle m(k)$ multiplied by the output scaling factor $\alpha \cdot K_{\triangle m}$ gives the actual adjustment in the server allocation.

Figure 3.8(a) shows non-uniform membership functions for fuzzy control inputs $e(k)$ and $\triangle e(k)$. The fuzzification process assigns linguistic values to an input and determines their certainties (degree of membership) by using input membership functions. The certainty of linguistic value $m$ assigned to an input is denoted by $\mu(m)$. For example, if $e(k)$ is 1/8, the linguistic variable “$e(k)$” is assigned a value $PS$ and $\mu(PS)$ is 1, since the numeric value of $e(k)$ projects up to the peak of the membership function corresponding to linguistic value $PS$. If $\triangle e(k)$ is 1/16, “$\triangle e(k)$” is assigned values $ZE$ and $PS$ with certainties $\mu(ZE)$ and $\mu(PS)$ as 0.5, since the numeric value of $\triangle e(k)$ projects up to the middle of the overlapping part of the membership functions corresponding to linguistic values $ZE$ and $PS$. Based on the fuzzified inputs, the inference mechanism determines which rules should be applied to reach fuzzy conclusions. Let $\mu(m, n)$ denote the premise certainty of rule$(m, n)$ where $m$ and $n$ are membership functions. Following the standard max-min inference mechanism, the rules to be activated are defined as set of rule$(m, n)$ such that $\mu(m, n) > 0$, where $\mu(m, n) = \min(\mu(m), \mu(n))$. For example, if $e(k) = 1/8$ and $\triangle e(k) = 1/16$, the certainties of rule$(PS, ZE)$ and rule$(PS, PS)$ are $\mu(PS, ZE) = 0.5$ and $\mu(PS, PS) = 0.5$, respectively.

We use non-uniform membership functions for control inputs since it allows fine granularity control action near the equilibrium point. We have adopt an intuitive approach for determining the shape of membership functions. However, advanced filtering and machine learning techniques could potentially be applied to determine membership functions that will result in the best performance. We also use uniform membership functions illustrated Figure 3.8(b) for control inputs.
The defuzzification component combines the rules activated by the inference mechanism using the “center average” method and calculates the fuzzy controller output. The fuzzy rules activated by the Inference mechanism generate multiple fuzzy conclusions. In the center average method, the strength (certainty) of each fuzzy conclusion is multiplied by their respective output membership function center points and summed. The area thus obtained is divided by the sum of the certainties of each fuzzy conclusion and the result is taken as the numeric value of fuzzy controller output. Let $b(m, n)$ denote the center of the membership function of the result of $\text{rule}(m, n)$. The fuzzy control output is calculated as

$$\Delta m(k) = \frac{\sum_{m,n} b(m,n) \cdot \mu(m,n)}{\sum_{m,n} \mu(m,n)}.$$ 

The membership function of fuzzy control output determines the value of $b(m, n)$. In the example above, rule($PS,ZE$) gives the result $NS$ with $b(PS, ZE) = -1/4$ and rule($PS,PS$) gives the result $NL$ with $b(PS, PS) = -3/4$. Therefore, the fuzzy controller output $\Delta m(k)$ is $-1/2$. Figure 3.9 shows a large number of membership functions that we use for the control output $\Delta m(k)$ to increase the flexibility. A uniform input membership function shown in Figure 3.8(b) would give similar results only if fuzzy control inputs were much higher. For example, if $e(k) = 1/3$ and $\Delta e(k) = 1/6$, the fuzzy control output will be same as the case when non-uniform membership function is used. This shows that non-uniform membership function is more sensitive to small values of control inputs, which is usually found near the equilibrium point. Note that $\Delta U(k)$ in Figure 3.9 refers to the output of the fuzzy controller when it is integrated with the optimization model. The integrated approach is described in the next section.

Note that the mapping of the values of the inputs $e(k)$ and $\Delta e(k)$ could be outside of the range $[-1,1]$ due to arbitrarily large errors. The fuzzy controller functionality will not be affected because any mapped error greater than 1 or less than -1 will be fuzzified into rules $PL$ and $NL$ respectively as shown in Figure 3.8. To mitigate their performance impact, we can choose the scaling factors such that a large range of error and change in error can be accommodated.
3.2.3 The fuzzy control system: stability analysis

A control system is said to be stable if it would come to its equilibrium state after any external input, initial conditions or disturbances that have impressed the system. We analyze the stability of the proposed fuzzy control system by using Lyapunov’s direct method. It is a time domain method suitable for analyzing the stability of a non-linear system.

We apply an approach similar to one in [136] for finding a suitable Lyapunov function. We first define the difference between the equilibrium resource value and current one as

\[
\tilde{m}(k) = M(k) - m(k).
\]  

Equilibrium value refers to the resource allocation value for which the end-to-end response time reaches or is close to the target value. From (3.22), we get

\[
\tilde{m}(k + 1) = \tilde{m}(k) - \alpha K \Delta m \Delta m(k).
\]  

We choose \( V(\tilde{m}(k)) = \tilde{m}^2(k) \) as the Lyapunov function in the discrete time. It gives the distance of the allocated resources from its equilibrium value. By applying the Lyapunov stability theorem, we find the stability constraint as follows,

\[
|\Delta m(k)| < \frac{2\epsilon}{\alpha K \Delta m}.
\]  

The stability constraint is intuitive as it suggests that increasing the output scaling factor reduce the stability of the system.

3.2.4 Self-tuning controller to compensate for server-switching costs

An important practicability issue related to server provisioning in virtualized datacenters is the cost of reconfiguration and server switching. We compensate for server-switching costs by adding a self-tuning component to the proposed model-independent fuzzy controller, as shown in Figure 3.10. The output scaling factor \( \alpha \cdot K \Delta m \) is automatically adjusted by the scaling factor controller to the transient behavior of the end-to-end response time. The scaling factor controller works in similar way as the fuzzy controller. The output of the controller is the gain updating factor \( \alpha \), which allows on-line gain variation based on instantaneous behavior.
Figure 3.10: A self-tuning fuzzy controller.

Figure 3.11: The membership function for $\alpha$.

Figure 3.12: The fuzzy rule base for $\alpha$. 
of the system. The value of $\alpha$ needs to be positive to ensure the system stability. The membership function
of $\alpha$ is defined within the interval $[0,1]$, as shown in Figure 3.11.

Figure 3.12 shows the rule base for the scaling-factor controller. It is designed in association with the rule
base of the fuzzy controller. A few important considerations for the design are:

1. When the error is large but has the same sign as the change in error, $\alpha$ should be made very large to
prevent from further worsening the situation.

2. To avoid large overshoot and reduce the settling time, $\alpha$ is set at a small value when the error is big but
has the opposite sign as compared to the change in error. If the process delay is high, the controller may
not achieve expected output after allocating required number of servers, and hence, tries to overreact
by assigning too many servers in the next sampling period. This is compensated by adjusting the output
scaling factor to a small value.

3. When the error is small, there should be a wide variation of the gain depending on the process trend
to avoid large overshoot and undershoot. Overshoot refers to an output exceeding its final steady-state
value. Undershoot refers to an output falling below its final steady-state value. This rule will avoid an
excessive oscillation. For example, if the error has reached the set point but is moving away upward
from the set point rapidly, a large $\alpha$ will prevent an overshoot by preventing the upward motion more
severely. Similarly, if the error is small and self-correcting, a small $\alpha$ will prevent large undershoot.

4. To improve the controller performance under load disturbance, $\alpha$ should be sufficiently large around
the steady state. For example, if the error is small and has the same sign as a large change in error, $\alpha$
should be large to bring the system back to steady state within a short time.

5. At a steady state, when the error is small and the change in error is also small, $\alpha$ should be very small
to avoid oscillation problem around the equilibrium point.

### 3.2.5 Integration of Fuzzy Control and Optimization model

The proposed model-independent fuzzy controller promises to assure the percentile-based end-to-end re-
response time guarantee in multi-tier server clusters. However, the fuzzy controller alone does not assure the
efficiency of the server provisioning. In the optimization model given in Section 7.2.4, Eq. (7.3) describes how the average end-to-end response time of a request is bounded. If $\bar{U}$ equals to zero, the solution given by Eq. (3.10) will efficiently allocate sufficient virtual servers so that the average end-to-end response time of a request is no more than the bound $U$. As the queueing model based approach aims to provide the mean response time guarantee, the 90th-percentile end-to-end response time however will be greater than the bound.

Therefore, we integrate the fuzzy controller with the optimization model. This can be achieved by adjusting the value of parameter $\bar{U}$ based on the measured error, which is the end-to-end response time bound minus the measured 90th-percentile end-to-end response time.

The integrated approach uses the same fuzzy controller, except that the output of the controller is the adjustment value of parameter $\bar{U}$. During the $(k)$th sampling period, when the measured error of the 90th-percentile response time and the change in error are fed to the fuzzy controller as inputs, the output of the controller is the adjustment value of parameter $\bar{U}$, which is expressed as $\triangle \bar{U}(k)$. $\alpha K_{\Delta U}$ is the output scaling factor for controller’s performance tuning. As the result, the value of parameter $\bar{U}$ to be used for the optimization model during the $(k + 1)$th sampling period is

$$\bar{U}(k + 1) = \bar{U}(k) + \alpha K_{\Delta U} \triangle \bar{U}(k) = \int \alpha K_{\Delta U} \Delta U(k) dk.$$  \hspace{1cm} (3.25)

The rationale of the integrated approach is that increasing the value of $\bar{U}$ decreases the bound on the average end-to-end response time and thus reduces the measured 90th-percentile end-to-end response time error by allocating more servers, and vice versa. The advantage of controlling $\bar{U}$ is that the number of servers allocated to each tier is determined based on the optimization in each sampling interval of the fuzzy control process for efficient resource allocation.

### 3.3 Autonomic Performance Assurance with Self-Adaptive Neural Fuzzy Control

Internet workloads, which are often highly dynamic in nature [19, 113, 123], impose significant challenges on the autonomic performance management of applications hosted in a virtualized datacenter, as described in Section 1. To overcome these challenges, resource provisioning mechanisms need to be self-adaptive to highly dynamic workload variations. Self-adaptiveness is an important feature of autonomic performance
There are model-independent rule based fuzzy controllers that utilize heuristic knowledge for performance guarantee on Internet servers [69, 88, 136]. They use a set of pre-defined rules and fuzzy membership functions to perform control actions in the form of resource allocation adjustment. These controllers have some drawbacks. First, they are designed manually on trial and error basis, using heuristic control knowledge. There is no specific guideline for determining important design parameters such as the input scaling factors, the rule base and the fuzzy membership functions. Second, those design parameters are non-adaptive. They are not effective in the face of highly dynamic workloads. We conducted simulation of the rule based fuzzy control approach in the face of a highly dynamic workload that is illustrated in Figure 3.13. It is sudden step-changes based, similar to a workload scenario used in [123].

Simulation results in Figure 3.14 show significant deviation of the $95_{th}$-percentile end-to-end delay from its pre-specified target 1400 ms. We observe a relative delay deviation and temporal target violation of 47% and 38% respectively. Temporal target violation is a measure of percentage of times when the end-to-end
delay target is violated within the measuring time frame. The rule based fuzzy controller is unable to adapt itself to the highly dynamic workload since the rule base and fuzzy membership functions are fixed at the design time through trial and error. Moreover, its performance is sensitive to a statically chosen parameter, the input scaling factor. This problem exists for other rule-based fuzzy control approaches as well [88, 136]. For autonomic resource management and performance guarantee of Internet services, self-adaptive server provisioning is a critical and challenging issue.

We propose an autonomic server allocation approach based on a self-adaptive neural fuzzy control technique for percentile-based end-to-end delay guarantee in virtualized multi-tier server clusters. The self-adaptiveness of the proposed approach is due to its hybrid design based on control theoretical and machine learning techniques.

The main advantages of the proposed server allocation approach based on the neural fuzzy controller are as follows:

1. It is robust to highly dynamic workload intensity as well as characteristics and change in delay target due to its self-adaptive and self-learning capabilities.

2. It is model-independent. The parameter variations of the system performance and the unpredictability of dynamic workloads do not affect the validity and effectiveness of the proposed server allocation approach.

3. It is capable of automatically constructing the control structure and adapting control parameters through fast online learning. The controller executes resource allocation adjustment and learns to improve its performance simultaneously.

4. Unlike other supervised machine learning techniques, it does not require off-line training. Avoiding off-line training saves significant amount of time and efforts required to collect a large set of representative training data and to train the system.

Figure 3.15 shows the block diagram of the dynamic server provisioning approach with a self-adaptive neural fuzzy control. The task of the controller is to adjust server provisioning on multi-tier clusters in order to bound the $95\%$-percentile end-to-end delay $T_d$ to a specified target $T_{ref}$. The controller has two inputs;
Figure 3.15: Block diagram of a self-adaptive neural fuzzy control.

error denoted as $e(k)$ and change in error denoted as $\Delta e(k)$. Error is the difference between the target and the measured value of the end-to-end delay in the $k^{th}$ sampling period, which is target delay minus measured delay. The output of the controller is the resource adjustment $\Delta m(k)$ for the next sampling period.

The decomposition of server resource adjustment $\Delta m(k)$ to the multiple tiers is performed in proportion to the per-tier delay observed from the controlled system. We choose per-tier delay rather than per-tier utilization because it directly affects the end-to-end response time. As the number of servers to be allocated to each tier can be a real value by the decomposition approach, there are two options. The first is at fine granularity. That is, a server capacity can be reconfigured according to the real value at run time with the modern server virtualization technology. However, it requires that the hosting physical machine has available resources for the virtual server resizing. Otherwise, it may require server migrations, which are often costly [52, 56]. The second is at the coarse granularity of a whole server. It uses the nearest integer value and the minimum is one. This option is feasible when it can find a physical machine for the server replication. In a datacenter, the administrator can use either option or the hybrid. As related works in [69, 73, 123], we adopt the second option in this work. But the neural fuzzy control based approach supports either option and the hybrid as well.

The neural fuzzy controller is designed to tolerate 5% error within the end-to-end delay target. As long as the error in delay is within the tolerance bound, it stops further control actions. The controller uses an online learning algorithm to automatically construct its structure and adapt its parameters. Online learning is a category of machine learning techniques, which learns one instance at a time from each data collected from the system. This makes it suitable to be deployed at run time.
Table 3.1: Notation Summary.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{ref}$</td>
<td>The percentile-based end-to-end delay target</td>
</tr>
<tr>
<td>$T_d$</td>
<td>The measured percentile-based end-to-end delay</td>
</tr>
<tr>
<td>$k$</td>
<td>Index of the control sampling interval</td>
</tr>
<tr>
<td>$i$</td>
<td>Index of the input variable in a fuzzy logic rule</td>
</tr>
<tr>
<td>$j$</td>
<td>Index of the fuzzy membership function</td>
</tr>
<tr>
<td>$r$</td>
<td>Index of the fuzzy logic rule</td>
</tr>
<tr>
<td>$e(k)$</td>
<td>Error for the control input</td>
</tr>
<tr>
<td>$\Delta e(k)$</td>
<td>Change in error for the control input</td>
</tr>
<tr>
<td>$\Delta m(k)$</td>
<td>The resource adjustment for the server system</td>
</tr>
</tbody>
</table>

3.3.1 Neural Fuzzy Controller

We design the neural fuzzy controller using a general four-layer fuzzy neural network as shown in Figure 3.16. The various layers of the neural network and their interconnections provide the functionality of membership functions and rule base of a fuzzy controller. Unlike a rule based fuzzy controller, the membership functions and rules dynamically construct and adapt themselves as the neural network grows and learns. Hence, the proposed controller is robust to highly dynamic workload variation. Table 3 summarizes important notations.

The fuzzy neural network adopts fuzzy logic rules as follows:

$$R_r: \text{IF } x_1 \text{ is } A_1^j \ldots \text{ and } x_n \text{ is } A_n^j, \text{ THEN } y \text{ is } b_r$$
where \( R_r \) is the \( r_{th} \) fuzzy logic rule, \( x_i \) is an input, either to be \( e(k) \) or \( \Delta e(k) \), and \( y \) is the rule’s output. \( A_j^i \) is the \( j_{th} \) linguistic term associated with the \( i_{th} \) input variable in the precondition part of the fuzzy logic rule \( R_r \). Linguistic terms are fuzzy values such as “positive small”, “negative large”, etc. They describe the input variables with some degree of certainty, determined by their membership functions \( u_{A_j^i} \). The consequent part or outcome of the rule \( R_r \) is denoted as \( b_r \). Each rule contributes to the controller output, denoted as \( \Delta m(k) \) according to its firing strength.

The functions of the nodes in each layer are as follows:

**Layer (1):** Each node in this layer corresponds to one input variable. These nodes only pass the input signal to the next layer. The proposed neural fuzzy controller has two input nodes corresponding to \( e(k) \) and \( \Delta e(k) \). The net input and net output for the \( i_{th} \) node are:

\[
net_i^{(1)} = x_i, \quad y_i^{(1)} = f_i^{(1)}(net_i^{(1)}) = net_i^{(1)}
\]  
(3.26)

**Layer (2):** Each node in this layer acts as a linguistic term assigned to one of the input variables in layer (1). These nodes use their membership functions to determine the degree to which an input value belongs to a fuzzy set. A Gaussian function is adopted as the membership function. For the \( j_{th} \) fuzzy membership node, the net input and net output are:

\[
net_{ji}^{(2)} = -\frac{(x_i - m_{ji})^2}{\sigma_{ji}^2}
\]  
(3.27)

\[
y_{ji}^{(2)} = u_{A_j^i} = f_j^{(2)}(net_{ji}^{(2)}) = \exp(net_{ji}^{(2)})
\]  
(3.28)

Here, \( m_{ji} \) and \( \sigma_{ji} \) are the mean and standard deviation of a Gaussian function of the \( j_{th} \) linguistic term associated with \( i_{th} \) input variable. They determine the position and shape of the input membership functions. As shown in Figure 3.16, let a node represent a linguistic term \( A_1^1 \) for the input variable \( x_1 \), which is \( e(k) \). Assume that its membership function \( u_{A_1^1} \) has a mean \( m_{11} \) and standard deviation \( \sigma_{11} \) of -50 and 20 respectively. \( A_1^1 \) is a fuzzy value such as “negative small”, “negative large”, etc. that corresponds to the numeric value of -50 with absolute certainty. The degree of certainty is calculated by using the membership function \( u_{A_1^1} \). If the measured error in the percentile-based end-to-end delay \( e(k) \) is -40, the output of the node will be 0.77 from Eq. (3.27). Similarly, let another node represent a linguistic term \( A_2^1 \) for the input variable \( x_2 \), which is \( \Delta e(k) \). Assume that its membership function \( u_{A_2^1} \) has a mean and standard deviation of -30 and 10
respectively. If the change in error $\Delta e(k)$ is -30, the output of the node is 1.

Layer (3): Each node in this layer represents the precondition part of one fuzzy logic rule. Each node multiplies the incoming signals and outputs the product result, i.e., the firing strength of a rule. For the $r^{th}$ fuzzy rule node,

$$net_r^{(3)} = u_{A_1} \cdot u_{A_2} \cdots u_{A_n}$$

$$y_r^{(3)} = u_r = f_r^{(3)}(net_r^{(3)}) = net_r^{(3)}$$

(3.29)

(3.30)

where $n$ is the number of input variables. The outputs of Layer (2) will be the inputs to this layer. From the previous example, the inputs to a node in this layer are 0.77 and 1. As a result, the net input and the net output will be 0.77.

Layer (4): This layer acts a defuzzifier. It converts fuzzy conclusions from Layer (3) into numeric output in terms of resource adjustment $\Delta m(k)$. The single node in this layer sums all incoming signals to obtain the final inferred result. The net input and net output are:

$$net^{(4)} = \sum_{r=1}^{M} w_r \cdot u_r$$

$$y^{(4)} = f^{(4)}(net^{(4)}) = net^{(4)}$$

(3.31)

(3.32)

where the link weight $w_r$ is the output action strength associated with the $r^{th}$ rule and $y^{(4)}$ is the output of the neural fuzzy controller. For example, if the link weight $w_r$ is 3, the output $\Delta m(k)$ of this layer will be 2.31 since $u_r$ is 0.77. This result is intuitive because negative values of $e(k)$ and $\Delta e(k)$ imply that the percentile-based end-to-end delay is greater than its target and the situation is further worsening. Thus, the neural fuzzy controller allocates more servers to reduce the error. The magnitude of resource adjustment depends on various parameters and interconnections of the neural fuzzy controller, which are determined and adapted dynamically as described in the next section.

Note that the control framework can be extended for integration with the utility computing paradigm when needed. The error term $e(k)$ in the controller can be replaced by a utility function that captures the cost-benefit tradeoff of server allocations.
3.3.2 Online Learning of Neural Fuzzy Controller

The neural fuzzy controller combines fuzzy logic’s reasoning with the learning capabilities of an artificial neural network. It is capable of automatically learning its structure and parameters using online request response time measured from a live system. Initially, there are only input and output nodes in the neural network. The membership and the rule nodes are generated dynamically through the structure and parameter learning processes are described as follows.

3.3.2.1 Structure Learning Phase

For each input node in layer (1), the structure learning technique decides to add a new node in layer (2) and the associated rule node in layer (3), if all the existing rule nodes have firing strength smaller than a certain degree threshold. Low firing strength of rule nodes imply that the input data pattern of error and change in error is not recognized by the existing neural network. Hence, the neural network needs to grow. We use a decaying degree threshold to limit the size of the neural network. The new nodes at layer (2) have membership functions with a mean \( m_{ji}^{\text{new}} \) equal to the input \( x_i \) and standard deviation \( \sigma_{ji}^{\text{new}} \) equal to a pre-specified or a randomly generated value.

To avoid the newly generated membership function being too similar to the existing one, the similarities between the new membership function and the existing ones must be checked. We use the similarity measure proposed in [83] to check the similarity of two membership functions. Suppose \( u_A(x) \) and \( u_B(x) \) are two Gaussian membership functions with means \( m_A, m_B \) and standard deviations \( \sigma_A, \sigma_B \) respectively. Then the similarity measure \( E(A, B) \) is given by:

\[
E(A, B) = \frac{|A \cap B|}{\sigma_A \sqrt{\pi} + \sigma_B \sqrt{\pi} - |A \cap B|}. \tag{3.33}
\]

Without loss of generality, assuming \( m_A \geq m_B \),

\[
|A \cap B| = \frac{1}{2} \frac{\text{h}^2(m_B - m_A + \sqrt{\pi}(\sigma_A + \sigma_B))}{\sqrt{\pi}(\sigma_A + \sigma_B)} \tag{3.34}
\]

\[
+ \frac{1}{2} \frac{\text{h}^2(m_B - m_A + \sqrt{\pi}(\sigma_A - \sigma_B))}{\sqrt{\pi}(\sigma_B - \sigma_A)} \tag{3.35}
\]

\[
+ \frac{1}{2} \frac{\text{h}^2(m_B - m_A - \sqrt{\pi}(\sigma_A - \sigma_B))}{\sqrt{\pi}(\sigma_A - \sigma_B)}. \tag{3.36}
\]
where \( h(x) = \max(0, x) \). In the case of scenario \( \sigma_A = \sigma_B \),

\[
|A \cap B| = \frac{1}{2} h^2(m_B - m_A + \sqrt{\pi} (\sigma_A + \sigma_B)) \frac{1}{\sqrt{\pi}(\sigma_A + \sigma_B)}.
\] (3.37)

If the similarity measure between the new membership function and the existing ones corresponding to either input variable is less than a pre-specified value, both new membership functions are adopted. Since the generation of membership functions in layer (2) corresponds to the generation of a new fuzzy rule, the link weight, \( w_{new} \), associated with a new fuzzy rule has to be decided. Generally, the link weight is initialized with a random or pre-specified value. The neural fuzzy controller applies an intuitive understanding of the system behavior for initializing the link weight. If the measured error in delay \( e(k) \) is positive, the link weight is initialized with a randomly chosen small negative number and vice versa. For instance, a positive value of \( e(k) \) indicates that the observed delay is smaller than the target delay. It is reasonable to reduce the number of servers allocated to the system by initializing a negative \( w_{new} \). Thus, the controller takes corrective actions in the right direction from the beginning although the magnitude of the link weight has not been fully learned.

The structure learning phase dynamically determines proper input space fuzzy partitions and fuzzy logic rules, depending on the measured error and change in error in the percentile-based end-to-end delay. This is in contrast to a rule based fuzzy controller with heuristically designed rules, which uses input scaling factors and a fixed set of membership functions to statically determine the input space fuzzy partitions. Hence, the neural fuzzy controller performs consistently well for a wide range of error and delay targets.

### 3.3.2.2 Parameter Learning Phase

The parameter learning is used to adaptively modify the consequent part of existing fuzzy rules and the shape of membership functions to improve the controller’s performance in the face of highly dynamic workload variation. The goal of performance improvement is expressed as a problem of minimizing an energy function,

\[
E = \frac{1}{2} (T_{ref} - T_d)^2 = \frac{1}{2} (e(k))^2
\] (3.38)

where \( T_{ref} \) and \( T_d \) are the target and measured values of the percentile-based end-to-end delay. The learning algorithm recursively obtains a gradient vector in which each element is defined as the derivative of the energy function with respect to a parameter of the network. This is done by the chain rule method. The method is
referred to as the backpropagation learning rule as the gradient vector is calculated in the direction opposite to the flow of the output of each node. The backpropagation learning algorithm is described as follows.

Layer (4): The error term to be propagated is computed as

$$\delta^{(4)} = -\frac{\partial E}{\partial y^{(4)}} = \left[ -\frac{\partial E}{\partial e^{(k)}} \frac{\partial e^{(k)}}{\partial \hat{y}^{(4)}} \right] = \left[ -\frac{\partial E}{\partial e^{(k)}} \frac{\partial e^{(k)}}{\partial T_d} \frac{\partial T_d}{\partial \hat{y}^{(4)}} \right]$$

(3.39)

The link weight \(w_r\) is updated by the amount

$$\Delta w_r = -\eta_w \frac{\partial E}{\partial w_r} = \left[ -\eta_w \frac{\partial E}{\partial y^{(4)}} \frac{\partial y^{(4)}}{\partial \hat{y}^{(4)}} \frac{\partial \hat{y}^{(4)}}{\partial w_r} \right] = \eta_w \delta^{(4)} u_r$$

(3.40)

where \(\eta_w\) is the learning rate of the link weight. The weights in layer (4) are updated according to the following equation.

$$w_r(k + 1) = w_r(k) + \Delta w_r$$

(3.41)

where \(k\) denotes the current sampling interval. Thus, the output action strength or consequence associated with each fuzzy rule is adjusted in order to reduce the error in the percentile-based end-to-end delay.

Layer (3): Only the error term needs to be calculated and propagated in this layer. That is

$$\delta^{(3)}_r = -\frac{\partial E}{\partial \hat{y}^{(3)}} = \left[ -\frac{\partial E}{\partial \hat{y}^{(4)}} \frac{\partial \hat{y}^{(4)}}{\partial \hat{y}^{(3)}} \frac{\partial \hat{y}^{(3)}}{\partial \hat{y}^{(3)}} \right] = \delta^{(4)} w_r$$

(3.42)

Layer (2): The error term is computed as follows,

$$\delta^{(2)}_j = -\frac{\partial E}{\partial \hat{y}^{(2)}} = \left[ -\frac{\partial E}{\partial \hat{y}^{(3)}} \frac{\partial \hat{y}^{(3)}}{\partial \hat{y}^{(2)}} \frac{\partial \hat{y}^{(2)}}{\partial \hat{y}^{(2)}} \right] = \delta^{(3)} \hat{y}^{(3)} = \delta^{(3)} u_r$$

(3.43)

The update law for \(m_{ji}\) is

$$\Delta m_{ji} = -\eta_m \frac{\partial E}{\partial m_{ji}} = 2\eta_m \delta^{(2)}_j (x_i - m_{ji}) \frac{1}{(\sigma_{ji})^2}$$

(3.44)

The update law for \(\sigma_{ji}\) is calculated as

$$\Delta \sigma_{ji} = -\eta_\sigma \frac{\partial E}{\partial \sigma_{ji}} = 2\eta_\sigma \delta^{(2)}_j \frac{(x_i - m_{ji})^2}{(\sigma_{ji})^3}$$

(3.45)

where \(\eta_m\) and \(\eta_\sigma\) are the learning-rate parameters of the mean and the standard deviation of the Gaussian function, respectively. The mean and standard deviation of the membership functions in this layer are updated as following.

$$m_{ji}(k + 1) = m_{ji}(k) + \Delta m_{ji}$$

(3.46)

$$\sigma_{ji}(k + 1) = \sigma_{ji}(k) + \Delta \sigma_{ji}$$

(3.47)
Thus, the position and the shape of the membership functions are adjusted dynamically. The exact calculation of the Jacobian of the system, $\partial T_d/\partial y^{(4)}$ in Eq. (3.39), cannot be determined due to the unknown dynamics of the multi-tier server clusters. To overcome this problem, we apply a delta adaptation law proposed in [84] as follows,

$$\delta^{(4)} = e(k) + \Delta e(k)$$  \hspace{1cm} (3.48)

The proof of the convergence of the neural fuzzy controller using Eq. (3.48) is similar to that in [84] and is omitted here.

### 3.4 Evaluation

#### 3.4.1 Resource Allocation Optimization with Performance Guarantee

We build a simulator for a typical three-tier web cluster and conduct extensive simulations to evaluate the new server provisioning approach based on queueing models. A synthetic session-based workload is generated according to a customer behavior model in [95] having an exponentially distributed think time with a mean of 5 seconds. We use bounded pareto distributions that are representatives for modeling the service time distribution in Internet applications [47, 137]. We adopt two such sets of characteristics as shown in Table 3.2. $E[X_i]$ and $E[X_i^2]$ are the first moment and second moment of the service time distribution $X_i$ at tier $i$, respectively. During the simulation, the end-to-end response time was measured periodically with a sampling interval of 1 minute. Each result reported is an average of 100 runs. As a case study, we assume that each virtual server follows FCFS scheduling discipline.
3.4.1.1 Impact of the optimization on resource allocation

First, we evaluate the resource allocation efficiency and end-to-end response time guarantee of our optimization-based server provisioning approach. For comparison, we experimented the per-tier decomposition based approach in [123] that sets the per-tier average response time targets to be 10%, 50%, and 40% of the end-to-end target, respectively. We also experimented a balanced decomposition based approach that sets each tier’s average response time target to be 1/3 of the end-to-end response time target. We set the target average response time to be 300 msec for the characteristics A and B, respectively.

Figure 3.17 shows the total number of servers required by three approaches for the average end-to-end response time guarantee at varying session arrival rate. It shows that the optimization-based approach uses the minimum number of servers for the response time guarantee. It also shows that the impact of the delay decomposition on the server usage is dependent on the workload characteristics. Figure 3.17(a) shows that by using the characteristic A, the balanced approach uses fewer servers than the 10%-50%-40% approach. Figure 3.17(b) shows that by using the characteristic B, the 10%-50%-40% approach uses fewer servers than the balanced approach. Overall, the experimental results show that the optimization-based approach can reduce the total number of servers allocated by about 20% compared to the 10%-50%-40% decomposition approach, and by about 25% compared to the balanced decomposition approach. Figures 3.18 and 3.19 show the number of servers at each tier due to three approaches for the average end-to-end response time guarantee due to the use of characteristic A and B respectively.
(a) By 10%-50%-40% approach.  
(b) By balanced approach.  
(c) By optimization-based approach.

Figure 3.18: The server allocation at each tier using the workload characteristic A.

(a) By 10%-50%-40% approach.  
(b) By balanced approach.  
(c) By optimization-based approach.

Figure 3.19: The server allocation at each tier using the workload characteristic B.

(a) With workload A.  
(b) With workload B.

Figure 3.20: End-to-end response time due to the optimization-based allocation.
3.4.1.2 Impact of the optimization on performance assurance

Next, we study the impact of optimization based server provisioning on the end-to-end response time of multi-tier application. Figure 3.20 shows the average end-to-end response time with its 90th and 10th percentiles at varying session arrival rate due to the optimization based approach. We observe that the average end-to-end response time is within the SLA target. However, there is no control on the 90th-percentile end-to-end response time. This is due to the fact that queueing models can only capture mean based performance.

3.4.1.3 Percentile-based response time guarantee with fuzzy controller

We conduct experiments to evaluate the effectiveness of the proposed fuzzy controller in achieving percentile-based response time guarantee. The experiment is performed with a workload of 1600 sessions per minute. The workload characteristic A is adopted with the target 90th-percentile end-to-end response time of 200 msec. We also investigate the impact of using non-uniform and uniform membership functions for the fuzzy control design.

Figures 3.21(a) and 3.21(b) show the average end-to-end response time with its 90th and 10th percentiles.
at different intervals due to the use of uniform and non-uniform membership functions, respectively. Both approaches can achieve a steady-state end-to-end response time guarantee. We later demonstrate that a self-tuning control component can significantly speed up the convergence. Figure 3.21(c) shows the performance comparison in terms of the standard deviation of the end-to-end response time from the steady-state value and the convergence rate of the end-to-end response time to the steady-state value. The convergence rate is the number of sampling intervals that the fuzzy controller takes to reach the steady state. The results show that the control using non-uniform membership functions achieves slightly better performance. It should be noted that the convergence rate also depends on the amount of tolerance allowed by the system in terms of steady state error. Increasing the tolerance may result in faster convergence.

The improvement of the server usage due to the use of non-uniform membership functions is much more significant. Figures 3.21(d) and 3.21(e) show the number of servers allocated to each tier due to the use of different membership functions. Figure 3.21(f) shows the improvement in the percentage at a few intervals. They show that the use of non-uniform membership functions can reach the steady-state value earlier and reduce the usage of servers for the end-to-end response time guarantee. Thus, server allocations and switching
times are significantly reduced. Using the non-uniform membership function may allocate more servers at the database tier, but the total number of servers allocated is smaller. The improvement is due to the fact that the non-uniform membership functions allow finer control nearby the equilibrium point that significantly mitigates over-provisioning of servers nearby the equilibrium point. Figure 3.22 shows similar results due to the workload characteristic B.

We establish the fact that the shape of fuzzy membership functions can have significant impact on the effectiveness and efficiency of the proposed model-independent fuzzy control based resource provisioning technique. In the next section, we explore the feasibility of applying hybrid of machine learning and control techniques to achieve a self-adaptive fuzzy controller whose membership functions as well as rule base automatically adapt themselves in response to the system performance and workload variations.

3.4.1.4 Impact of the self-tuning controller on performance

To evaluate the impact of the self-tuning capability on the controller performance, we simulate the instability of end-to-end response time during the addition of servers in a tier. Figure 3.23 shows the performance difference of the fuzzy control system with and without the self-tuning capability. The experiment was performed
with a workload of 1600 sessions per minute, the characteristic A and the target 90th-percentile end-to-end response time of 200 msec. From 3.23(a) and (b), we observe that the target end-to-end response time is reached much earlier by the use of the self-tuning fuzzy controller that reduces unnecessary oscillations during the server allocation processes. It leads to a smaller number of server switchings as compared to the fuzzy control system without the self-tuning capability. Figure 3.23(c) shows that the self-tuning capability can significantly speed up the convergence rate for the end-to-end response time guarantee and slightly reduce the response time deviation. Moreover, the total number of servers allocated at the steady state is smaller by the use of the self-tuning fuzzy controller, as shown in Figures 3.23(d)-(f). It is due to the fact that the control process is more finely and adaptively tuned with the self-tuning capability. Note that the impact of the self-tuning capability on the server usage at different tiers is different. Figure 3.24 shows similar results due to workload characteristic B.

3.4.1.5 Impact of optimization-fuzzy integration on resource allocation

In this section, we study the impact of the optimization-fuzzy integration on performance when subjected to a dynamic workload. With the integration of our optimization model with fuzzy controller, optimization
is performed at every sampling interval of fuzzy control action. This is done by adjusting the controllable parameter $\bar{U}$ as discussed in Section 3.2.5. In the experiment, the session arrival rate is at 800 sessions per minute in the first 11 intervals and it changes to 1600 sessions per minute at the interval 12. The workload characteristic A is used with the target 90th-percentile end-to-end response time of 200 msec.

Figures 3.25(a)-(c) show that the approach with the integration and the approach without the integration have a similar deviation of the end-to-end response time and a similar convergence rate to the steady state. This is due to the effectiveness of the fuzzy controller in end-to-end response time guarantee. During the intervals 12 and 13, there are spikes in the end-to-end response time by both approaches due to the doubled session arrival rate. Results show that the self-tuning fuzzy controller is able to eventually adjust the server allocations and assure the 90th-percentile end-to-end response time guarantee.

Figures 3.25(d)-(f) show the efficiency impact of the optimization-fuzzy integration on the number of servers allocated to the multi-tier system. At the final steady state (intervals 16 to 30), the approach with the integration and the approach without the integration use total 175 and 220 servers respectively. Note that they have very little difference in the end-to-end response time. This is due to the fact that without the integration,
the allocated servers are not distributed effectively to the individual tiers of a multi-tier system according to the workload characteristics. Though the impact on the server usage at different tiers is different, the integrated approach can significantly reduce the total number of servers needed for the end-to-end response time guarantee. The figures show that by applying the optimization-fuzzy integration, the multi-tier system uses about 20% fewer servers at the final steady state. It demonstrates the efficiency merit due to the integration. We also conducted the experiment with the workload characteristic B. We found similar resource allocation efficiency impact of the optimization-fuzzy integration.

We observe that the server allocation efficiency merit due to the optimization-fuzzy integration by processor sharing is 20%, which is less than that due to the optimization-fuzzy integration by FCFS under the same experimental setup (refer to section 5.4). A fuzzy controller without integration of the optimization model distributes servers at various tiers according to the measured average resource demand at each tier. The average resource demand has a significant contribution to the optimization result obtained for processor sharing model of virtual servers. The optimization result with the processor sharing does not contain the second moment of service time distribution, unlike in the FCFS case. Thus, we believe that the distribution of servers to various tiers according to the average resource demand is closer to the optimal distribution in case of processor sharing as compared to the FCFS case. Hence, the efficiency merit of the optimization-fuzzy integration by processor sharing is less than that by FCFS. Nevertheless, the efficiency merit due to the optimization-fuzzy integration is significant in either case.

As a summary, the designed fuzzy controller is effective in assuring end-to-end performance guarantee. The results highlight the model independence as a key strength of the fuzzy control based server provisioning approach. They also demonstrate the efficiency merit due to the integration of the fuzzy control and the resource allocation optimization.

### 3.4.2 Autonomic Performance Assurance with Self-Adaptive Neural Fuzzy Control

We evaluate the server provisioning approach based on the self-adaptive neural fuzzy control in a typical three-tier server cluster with extensive simulations. In simulations, as others in [19] we assume that the database tier can be replicated on-demand as it employs a shared architecture. The controlled multi-tier
Table 3.3: Workload characteristics A.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>WebTier</th>
<th>AppTier</th>
<th>DBTier</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>20 ms</td>
<td>294 ms</td>
<td>254 ms</td>
</tr>
<tr>
<td>$\sigma_i^2$</td>
<td>848</td>
<td>2304</td>
<td>1876</td>
</tr>
</tbody>
</table>

system has a number of servers pre-allocated initially. For choosing the initial server allocation, we first run a number of simulations for various server allocations to observe the percentile-based end-to-end delay in the face of a stationary workload of 12 requests per second. Then, we choose the server allocation that provided the end-to-end delay close to a target of 1500 ms. The setting is 1 web server, 6 application servers and 5 database servers. We use the same initial server allocation for all simulations to demonstrate that the dynamic provisioning approach based on the neural fuzzy controller is able to guarantee the percentile-based end-to-end delay target when the workload and delay targets vary dynamically. In practice, the initial server allocation in the controlled system may be decided by a datacenter administrator based on best practices.

The research in [96] proposed an interesting datacenter capacity planning approach that exploits statistical multiplexing among the workload patterns of multiple virtual machines. The research in the initial datacenter capacity planning is complementary to our research in dynamic server provisioning.

We generate a synthetic workload using Pareto distributions of request inter-arrival time and service time, following a $G/G/1$ FCFS queueing model. Pareto distribution representing a heavy-tailed traffic has close resemblance to real Internet traffic that is bursty in nature [69, 152]. Although the workload is generated according to a specific model and requests are processed in a specific principle, our self-adaptive neural fuzzy controller itself does not make such assumptions. We choose the workload characteristics of a three-tier application reported in [123]. Table 3.2 gives the characteristics, in which $s_i$ and $\sigma_i^2$ are the average service time and the variance of service time distribution of requests at tier $i$, respectively. We apply dynamic request arrival rates with sudden step-changes similar to what used in [123] and with continuous changes similar to what used in [19]. Each representative result reported is an average of 100 runs.

We use two performance metrics, relative delay deviation as in [136] and target violation. Relative delay deviation is based on the square root mean of delay errors. It reflects the transient characteristics of a control system and measures how closely the percentile-based delay of requests follows a given target for $n$ sampling.
intervals. That is,

\[ R(e) = \sqrt{\frac{\sum_{k=1}^{n} e(k)^2}{n}} / T_{ref}. \]  

(3.49)

The relative deviation, however, does not differentiate whether the actual end-to-end delay is greater than or less than the target. It is indeed desirable that an actual end-to-end delay is less than the target. To measure the temporal violation of delay target, we define a metric of target violation

\[ T(v) = \frac{\sum_{k=1}^{n} v(k)}{n} \]  

(3.50)

where \( v(k) \) is one if the actual end-to-end delay is greater than the target \( T_{ref} \), and zero if it is less than or equal to \( T_{ref} \).

The neural network itself is empty initially. As a new node is added into the network, the neural fuzzy controller will choose the sign of the pre-configured link weight according to the observed error in delay. If the measured error in delay is positive, the link weight is initialized with a randomly chosen small negative number from the range of \([-6, -3]\] and vice versa. Choosing initial link weight in the range \([-6, -3]\) or \([3, 6]\) is to attenuate the resource allocation adjustments made by the neural fuzzy controller at its initial phase. The controller will allocate or deallocate at least three and at most six virtual servers each iteration. As a result, the controller is able to take corrective actions right from the beginning although the magnitude of the link weight has not been fully learned. The neural network grows as the values of error and change in error are calculated based on the measured end-to-end delay, and the magnitude of the link weight is updated by the fast online learning.

We initialize the neural fuzzy controller parameters as follows. When a new node is added in layer (2), its membership function has a standard deviation \( \sigma_{ji} \) equal to a pre-specified value 20. The standard deviation of a fuzzy membership function determines the range of input values that will trigger a particular fuzzy rule. Since neural network parameters associated with the newly added node are not yet learned, it needs to restrict the effect of the new node to a smaller range of input values. The learning rates \( \eta_w \), \( \eta_m \) and \( \eta_{sigma} \) are set to 0.8 for aggressive initial learning. As the neural network grows in size, the learning rate is gradually decreased in proportion to the number of rule nodes for fast convergence.

The end-to-end delay of the system is measured periodically at a control interval of 3 minutes. The choice of control interval depends on the trade-off between responsiveness of the controller and robustness
to measurement noise. The controller applies an exponential moving average method to further reduce the transient noise in delay measurement.

3.4.2.1 Effectiveness of Neural Fuzzy Control Approach

We evaluate the effectiveness of the new approach for performance guarantee under both dynamic and stationary workloads. First, we use a highly dynamic workload with sudden step-changes in request arrival rate as shown in Figure 3.13. We set the end-to-end delay bound to 1400 ms.

Figure 3.26(a) demonstrates the effectiveness of the neural fuzzy controller in assuring the $95_{th}$-percentile end-to-end delay guarantee. The extent of server allocation and de-allocation is crude at the beginning when the controller has gone through very few learning steps. The effect of adding or removing servers in such cases is usually not very good. Results show that during the first 24 minutes, the measured end-to-end delay oscillates significantly around the target delay of 1400 ms with the control actions. However in some cases, a single control action may be able to bring the end-to-end delay very close to the target depending upon the existing server allocation. For example, at time $60_{th}$ minute, adding 2 application servers and 2 database servers to the system is just enough to bring the end-to-end delay close to the target.

Figure 3.26(b) shows the corresponding server allocations. The controller stops the control action as long as the error in the end-to-end delay is within the 5% delay tolerance bound. This improves the system stability. For example, during $30_{th}$ minute and $45_{th}$ minute, there is no change in the server allocation. Note that the number of servers allocated to the web tier remains the same. This is due to the workload characteristics used
in Table 3.2. The web tier has relatively small resource demand compared to the application and database tiers. Hence, the controller allocates more servers to the application tier and database tier.

Figures 3.27(a) and (b) show the per-tier delay and server utilization under the highly dynamic workload. The per-tier delay of requests as well as server utilization vary across tiers mainly due to the characteristics of per-tier resource demands of the given workload as shown in Table 3.2.

Overall, experimental results show that the new server provisioning approach achieves a small relative delay deviation of 14% and target violation of 17% respectively. This is a significant improvement from the performance of the rule based fuzzy controller for the same workload scenario in Figure 3.14, where the relative delay deviation and the target violation are 47% and 38% respectively.

The neural fuzzy controller is robust to highly dynamic workload variation due to its self-adaptive capability. There are a few spikes in the end-to-end delay due to sudden changes in the applied workload.
Figure 3.29: Per-tier performance of neural fuzzy control for a stationary workload.

However, it achieves the delay guarantee in a very responsive manner, usually in few intervals after there is a sudden step-change in the request arrival rate.

We also apply a stationary workload with an average request arrival rate of 12 requests per second. Figures 3.28(a) and 3.28(b) show the end-to-end delay variation and changes in server allocation. There is an overshooting in the number of application and database servers allocated at the beginning. It is due to the fact that the neural fuzzy controller is empty at the beginning. It learns how to make control decisions at run time. As the learning process proceeds, the control behavior improves over time. Results show that the neural fuzzy controller is able to guarantee the 95\textsuperscript{th}-percentile end-to-end delay target of 1000 ms within a couple of sampling intervals, in spite of the fact that the controller starts its operation with an empty structure. This is due to its capability to self-construct its structure and to adjust its parameters through the fast online learning algorithm.

Figures 3.29(a) and 3.29(b) show the per-tier delay and server utilization under the stationary workload. The per-tier delay of requests as well as server utilization vary across tiers mainly due to the characteristics of per-tier resource demands of the given workload as shown in Table 3.2. More stable results compared to those in Figures 3.27(a) and 3.27(b) are due to the fact that the workload is stationary rather than highly dynamic.
Figure 3.30: 95th-Percentile end-to-end delay assurance for a dynamic step-change workload (target 1500 ms).

(a) with RBFC.  
(b) with NFC.  
(c) performance comparison.

Figure 3.31: Performance comparison for various delay targets with dynamic step-change workload.

(a) relative delay deviation.  
(b) temporal violation.  
(c) performance difference.

3.4.2.2 Comparison With Rule Based Fuzzy Controllers

A rule based fuzzy controller has shown its merits in achieving performance assurance through model-independent resource allocation and dynamic output scaling factor tuning [69, 73]. However, it shows inconsistent delay guarantee and significantly more target violations in case of highly dynamic workloads. That is mainly due to the fact that the rule based fuzzy controller applies statically chosen input scaling factor, rule base and membership functions that are manually tuned for a particular workload and a delay target. In this section, we compare the performance of our neural fuzzy controller (NFC) with a rule based fuzzy controller (RBFC) used in [69, 73]. We choose an input scaling factor of 1/500 as it shows good performance under a stationary workload for the rule based fuzzy controller. For quantitative comparison, we take the performance of NFC as a baseline and define the performance difference between NFC and RBFC as

\[
PD_{\text{deviation}} = \frac{R(e)_{\text{RBFC}} - R(e)_{\text{NFC}}}{R(e)_{\text{NFC}}} 
\]

(3.51)

\[
PD_{\text{violation}} = \frac{T(v)_{\text{RBFC}} - T(v)_{\text{NFC}}}{T(v)_{\text{NFC}}} 
\]

(3.52)
If PD is positive, the NFC has better performance than RBFC and vice versa.

**Dynamic Workload with Sudden Step-Change Request Arrival Rate**  First, we apply a dynamic workload with sudden step-changes in the request arrival rate as shown in Figure 3.13. Figure 3.30 shows that the self-adaptive neural fuzzy controller is more robust to the dynamic workload variation compared to the rule based fuzzy controller in assuring the $95_{th}$-percentile end-to-end delay guarantee (1500 ms). NFC outperforms RBFC by 76% and 33% in terms of the relative delay deviation and the target violation respectively. Its robustness to highly dynamic workloads is due to the self-adaptive and self-learning capabilities.

Next, we conduct sensitivity analysis of two controllers for various end-to-end delay targets. Figure 3.31(a) shows that the relative delay deviation tends to increase with the increase in the end-to-end delay target (from 800 ms to 1600 ms). This is due to the fact that larger delay targets require fewer servers for allocation, making it more difficult to achieve fine-grained control on the $95_{th}$-percentile end-to-end delay. As shown in Figure 3.31(b), the temporal target violation is small for medium range of delay targets between 1000 ms to 1400 ms. The delay targets higher than this range show more target violations due to a small number of servers involved in the control action. The targets in the lower range also result in larger target violations due to the fact that a controller takes more control intervals to reach very low delay targets. Compared to the rule based fuzzy controller, the new neural fuzzy controller consistently achieves less delay deviation and target violation for various delay targets. Figure 3.31(c) shows that NFC outperforms RBFC for all delay targets except 1100 ms. For that case, the rule based fuzzy controller has slightly better performance in terms of delay deviation because it is well suited for that particular delay target. On average, NFC performs better than RBFC by 32% and 59% in terms of delay deviation and target violation respectively. The main reason of the performance improvement is due to the fact that it adapts itself to accommodate various range of inputs instead of relying on statically chosen input scaling factor.

**Dynamic Workload with Continuous Change in Request Arrival Rate**  Our neural fuzzy control based server provisioning approach is applicable to any type of workload and delay guarantee metric due to its self-adaptive capability. We illustrate this by experimenting on a scenario with continuously changing load similar to what used in [19], in which the workload request arrival rate variation follows a sinusoid (sine) function.
Figure 3.32: A continuously changing dynamic workload for a three-tier Internet service.

(a) with RBFC.  
(b) with NFC.  
(c) performance comparison.

Figure 3.33: Median end-to-end delay for a continuously changing workload (target 1000 ms).

(a) relative delay deviation.  
(b) temporal violation.  
(c) performance difference.

Figure 3.34: Performance comparison for various delay targets with continuously changing workload.
Figure 3.32 illustrates the workload scenario. As a case study, we aim to assure the median end-to-end delay guarantee instead of 95\textsuperscript{th}-percentile based guarantee. This experiment is to demonstrate the capability of the neural fuzzy control in assuring other percentile-based end-to-end delay guarantee.

As shown in Figure 3.33, the median end-to-end delay is mostly above the target of 1000 ms as the workload continuously increases till the time 60 minutes. Both controllers allocate more servers to reduce the median end-to-end delay. After time 60 min, the median end-to-end delay is mostly below the target as the workload continuously decreases. In this case, both controllers de-allocate servers to bring the end-to-end delay close to the target. Throughout the experiment, the self-adaptive NFC is able to keep the median end-to-end delay closer to the target as compared to RBFC. NFC outperforms RBFC by 85\% and 14\% in terms of the relative delay deviation and the target violation respectively.

Next, we conduct sensitivity analysis of two controllers for various end-to-end delay targets. Figures 3.34(a), (b) and (c) show that the neural fuzzy controller consistently outperforms the rule based fuzzy controller in terms of the relative delay deviation and target violation for various delay targets. Similar to the results observed in case of the sudden step-change workload, the performance difference varies depending on the range of delay targets for which the rule based fuzzy controller is well tuned. For example, the performance differences in relative delay deviation and temporal target violation range from 17\% to 42\% and 0\% to 17\% respectively for delay targets between 1300 ms and 1600 ms. For delay targets outside this range, the performance difference is much larger. It is 48\% to 165\% for relative delay deviation and 14\% to 73\% for temporal target violation. On average, NFC outperforms RBFC by 68\% and 26\% in terms of delay deviation and target violation respectively.

### 3.4.2.3 Impact of Input Scaling Factor on Controller’s Self Adaptivity

We now study the impact of the input scaling factor on the performance of the rule based fuzzy controller, and compare its performance with our neural fuzzy controller. For RBFC, through trial-and-error we observed a certain ratio between the input scaling factors for $\Delta e(k)$ and $e(k)$ that gave the best performance. We found this ratio is about 1:3 for the given workloads. For RBFC in the experiments, the input scaling factor for $\Delta e(k)$ is chosen as one-third of the input scaling factor for $e(k)$. For example, when the input scaling factor
(a) relative delay deviation.  \hspace{1cm}  (b) temporal violation.  \hspace{1cm}  (c) performance difference.

Figure 3.35: Performance comparison for various input scaling factors with delay target 1400 ms in case of dynamic step-change workload.

(a) relative delay deviation.  \hspace{1cm}  (b) temporal violation.  \hspace{1cm}  (c) performance difference.

Figure 3.36: Performance comparison for various input scaling factors with delay target 1000 ms in case of dynamic step-change workload.

(a) relative delay deviation.  \hspace{1cm}  (b) temporal violation.  \hspace{1cm}  (c) performance difference.

Figure 3.37: Performance comparison for various input scaling factors with delay target 1100 ms in case of continuously changing workload.
for $e(k)$ is $1/500$, it is $1/1500$ for $\Delta e(k)$. RBFC works well when it is relatively less sensitive to the change in error.

Figures 3.35 and 3.36 show their relative delay deviation, target violation and performance difference for end-to-end delay targets 1400 ms and 1000 ms respectively in case of the dynamic step-change workload. Results demonstrate that increasing the scaling factor may improve the performance of the rule based fuzzy controller for one delay target (1400 ms), but it may degrade the performance for another delay target (1000 ms). In both cases, the performance of the neural fuzzy controller is consistently better than the rule based control. We observe similar results in case of the continuously changing workload as shown in Figures 3.37 and 3.38.

When the input scaling factor is chosen as $1/100$, the rule based fuzzy controller shows very good performance in terms of relative delay deviation and temporal target violation for one delay target (1100 ms), but very poor performance for another delay target (1300 ms). For the target 1100 ms and input scaling factor $1/100$, its performance is similar to that of neural fuzzy controller. The neural fuzzy controller’s relative delay deviation is slightly better by 12% and the temporal target violation is slightly worse by 16%. For all other scaling factors, the neural fuzzy controller significantly outperforms the rule based fuzzy controller. For the target 1300 ms, the performance of two controllers is similar when the input scaling factor is chosen to be $1/1000$ but the performance of the neural fuzzy controller is significantly better for most of other scaling factors.

The rule based fuzzy controller is sensitive to the choice of the input scaling factor, which attempts to partition the input fuzzy space non-adaptively. For the optimal performance, it needs to be manually tuned
each time for different delay targets. In practice, a highly dynamic and realistic workload increases the possibility that the inputs to the fuzzy controller (i.e., error and change in error) may not fit into the input space fuzzy partitions as intended. Furthermore, the delay target can also change dynamically according to the service level agreement. Hence, there does not exist one single scaling factor that works best for different scenarios. Since the rule base and fuzzy membership functions are also fixed at the design time through trial and error, the rule based fuzzy controller is unable to adapt itself to a highly dynamic workload. Thus, we need a self-adaptive controller designed based on neural fuzzy control. The main reason behind the superior performance of the neural fuzzy controller in assuring the end-to-end delay guarantee is its self adaptivity and online learning capability as compared to trial and error based design of the rule based fuzzy controller.

### 3.4.2.4 Comparison with a PI controller under Varying Workload Characteristics

In this section, we evaluate the performance of our neural fuzzy control based server provisioning technique in case of varying workload characteristics. In practice, the workload characteristic of a multi-tier application varies according to its workload mix. For example, the service time distribution of browsing mix based workload requests is different than that of shopping mix based workload requests in an e-commerce application [113]. Classical proportional integral (PI) controllers have been widely used for admission control and performance assurance in Internet servers [60, 89]. We use the classical PI control technique as the baseline for comparison with the neural fuzzy controller. The control interval is 3 minutes in both controllers.

Since PI control is model-based, we first obtain the system performance model of our virtualized multi-tier system by using a standard system identification technique. We use the workload characteristic $A$ as shown in TABLE 3.3 and measure the end-to-end delay for various virtual server allocations. Based on the offline data collected from the system, we use the Least Squares Method (LSM) to estimate the parameters.
of the system model [132]. The parameters are given by Eq. (3.53).

\[ d'(k) = b_1 d'(k - 1) + c_1 m'(k - 1) \]  

(3.53)

where \( b_1 = -0.06829 \) and \( c_1 = -0.5149 \). The controlled variable in the system model is \( d'(k) = d(k) - d \) where \( d(k) \) is the end-to-end delay measured at sampling interval \( k \). Note that the end-to-end delay measurement can be percentile-based, mean or median based. The manipulated variable is \( m'(k) = m(k) - m \) where \( m(k) \) is the number of virtual servers allocated at sampling interval \( k \). \( d \) and \( m \) are the end-to-end delay and virtual server allocation corresponding to a chosen operating point that is used to linearize the system model. We choose operating points in the system by selecting the middle value of a typical range of virtual server allocation as \( m \), and then measure the resultant end-to-end delay as \( d \). Based on the system model, the PI controller is designed to achieve the desired control performance such as system stability and zero steady-state error. The transfer function of the designed PI controller is:

\[ F(z) = \frac{-0.34131(z + 1)}{(z - 1)} \]  

(3.54)

We compare the performance of server provisioning approach with our neural fuzzy controller and with the PI controller when the workload changes from characteristic A shown in TABLE 3.3 to characteristic B shown in TABLE 3.4. Figures 3.39(a) and (b) show the relative delay deviations for the two workload characteristics with stationary intensity and with dynamically varying intensity respectively. For stationary workload with characteristic A, our neural fuzzy controller shows similar performance as the PI controller. However, it significantly outperforms the PI controller for workload characteristic B. The PI controller is
designed for a system model that is identified by using workload characteristic A. Hence, it shows poor performance when the workload characteristics changes. On the other hand, our neural fuzzy controller is adaptive to varying workload characteristics. For highly dynamic workload, the neural fuzzy controller outperforms the PI controller for both workload characteristics A and B. This is because unlike the PI controller, it is adaptive to variation in workload intensity as well as characteristics.

As a summary, compared with the PI controller, there is an overall performance improvement of 61% by the use of the neural fuzzy controller. This value is the average of the performance difference metrics defined in Eq. (3.51) and Eq. (3.52), for the 4 scenarios - (1) stationary workload A, (2) stationary workload B, (3) dynamic workload A and (4) dynamic workload B.

3.4.2.5 A Case Study based on the Testbed Implementation

We conduct a feasibility study with performance evaluation of the proposed neural fuzzy control based server provisioning approach in a prototype datacenter, which consists of 12 HP ProLiant BL460C G6 blade server modules and a 40 TB HP EVA storage area network with 10 Gbps Ethernet and 8 Gbps Fibre/iSCSI dual channels. Each blade server is equipped with Intel Xeon E5530 2.4 GHz quad-core processor and 32 GB DDR3 memory. Virtualization of this cluster is enabled by VMWare’s vSphere 4.1 Enterprise edition. vSphere controls the disk space, memory, and CPU share (in MHz) allotted to the virtual machines, and also provides an application programming interface (API) to support the remote management of virtual machines. Our controller uses VMWare’s VIX API 1.10 as an actuator to dynamically instantiate or de-instantiate VMs on the hosts and to assign CPU shares to the virtual machines.

We have implemented a virtualized multi-tier server cluster architecture as shown in Figure 3.1. The database tier is not replicable in our testbed implementation. Each server in the multi-tier cluster is hosted inside a VMware virtual machine. The configuration of each virtual machine for the web and application tiers is 1 vCPU, 2 GB RAM and 15 GB hard disk space. We use 4 vCPUs, 2 GB RAM and 15 GB hard disk space for the database server to perform the case study where the database tier is not the bottleneck. Otherwise, the database tier server needs to be reconfigured with more resources or replicated. The guest operating system used is Ubuntu Linux version 10.04. Load balancers are used to distribute requests among virtual machines
at the web and application tiers. An Apache module, *mod_proxy_balancer*, is used for load balancing while taking into account session affinity.

The neural fuzzy controller interacts with the VM manager (VMM) through the vSphere Management API. To start a VM, the controller issues the VM start request using the *PowerOnVM_Task* method. It submits a virtual machine power-on task to the VMM. The controller obtains the task start time from the task information structure. When the power-on task has been completed, the controller obtains the task completion time. Using the task start and completion time, the controller calculates the time used for starting the VM. Similarly, it can obtain the time used for removing a VM. Our experiments show that adding or removing a VM in the testbed cluster takes approximately 7 seconds, which is quite small compared to the control interval of 3 minutes used for delay measurement.

As many related studies [74, 103, 123, 135], this work uses an open-source multi-tier application benchmark, RUBiS [110], in the experimental study. RUBiS implements the core functionality of an eBay like auction site: selling, browsing and bidding. It implements three types of user sessions, has nine tables in the database and defines 26 interactions that can be accessed from the clients Web browsers. The application contains a Java-based client that generates a session-oriented workload. RUBiS sessions have an average duration of 15 minutes and the average think time is 5 seconds. It defines two workload mixes: a browsing mix made up of only read-only interactions and a bidding mix that includes 15% read-write interactions. We configure the RUBiS clients to submit workloads of different mixes as well as workloads of time-varying intensity. Each RUBiS client also provides a sensor that measures the client-perceived QoS metrics such as average response time and throughput over a period of time. We modify the client to measure the percentile-based response time required by our experiments. Our implementation of RUBiS application is done with Apache 2.2.14, PHP 5.3.2 and MySQL 5.1 servers for the web, application and database tiers.

**End-to-end and Per-tier Delays by the Neural Fuzzy Controller** For performance evaluation, we apply a dynamic workload to our multi-tier virtualized server cluster as shown in Figure 5.2.5 (a). Initially, the workload consists of a bidding mix of 200 concurrent users. After 20 minutes we double the workload intensity to 400 concurrent users with browsing workload mix. Another 20 minutes later, we decrease the workload to 300 concurrent users. The reported results are from a single run. The $95_{th}$-percentile end-to-end
delay target is set to be 2 seconds.

Figure 5.2.5(a) shows that the self-adaptive neural fuzzy controller is able to guarantee the 95th-percentile delay target of 2 seconds within a few sampling intervals. The multi-tier system is initially provisioned with one virtual server at each tier. As the controller starts allocating virtual servers at the web and application tiers, it applies online learning to tune its neural network structure and parameters based on the measured percentile-based end-to-end delay of requests. We observe that the 95th-percentile delay approaches the target of 2 seconds within the first 15 minutes of the experiment as a result of dynamic server allocations. The oscillation of the delay around its target is mainly due to the fact that neural fuzzy controller needs to learn how to control the system by exploring different server allocations. As time progresses, the controller becomes more effective in achieving the end-to-end delay guarantee.

There is a spike in the measured end-to-end delay at time 20th minute due to the sudden increase in the workload intensity and the change in the workload mix. However, the neural fuzzy controller achieves the delay guarantee in a responsive manner, usually in three to four control intervals. Similarly, there is a sudden drop in the measured delay at time 40th minute due to the decrease in workload from 400 to 300 concurrent users. The neural fuzzy controller effectively removes virtual servers from different tiers to bring the end-to-end delay close to the target. Results show that the controller is robust to dynamically varying workload intensity as well as characteristics due to its self-adaptive capability.

Figure 5.2.5(b) shows the change in the allocation of virtual servers at various tiers. The controller allocates servers at individual tiers in proportion to the per-tier average delay measurement. Note that in the
testbed implementation, the server allocation adjustments are only distributed between the web and application tiers as the database tier is not replicated.

We also show the per-tier average delay and server utilization. Obtaining per-tier delays of a request is nontrivial. At the web tier (Apache) and the application tier (PHP), we can obtain the response time of each request from its log. However, at the database tier, the free community version MySQL 5.1 does not provide such a functionality for obtaining a query’s response time. That means, we can obtain the delay at the web tier, and the total delay but not the per-tier delay at the application and the database tiers. To timestamp each query issued from the application tier, we make an instrumentation of RUBiS application by modifying its PHP scripts. Thus, we are able to measure the per-tier delays. Note that the per-tier delay measured by this approach may not be perfectly accurate since it includes the time spent inside the application tier while constructing the query before sending it to database tier.

Figures 3.41 (a) and (b) show the per-tier average delay of requests and server utilization at various control intervals. The per-tier delay of requests as well as server utilization vary across tiers mainly due to the characteristics of per-tier resource demands of the given workload. In our experiment, the resource utilization at the database tier is small compared to the resource utilization at the web and application tiers mainly due to the high server capacity at the database server.

**Impact of Control Interval on Control Robustness and Delay Guarantee**

In the experiments reported so far, the end-to-end delay of the system is measured periodically at a control interval of 3 minutes. The choice
of control interval depends on the tradeoff between robustness to delay measurement noise and responsiveness of control actions. We evaluate the impact of choosing different control intervals on the end-to-end delay guarantee and control robustness to measurement noise.

Figures 3.42 (a) and (b) show the $95\%$-percentile delay achieved by the neural fuzzy controller when the control intervals are chosen as 1 minute and 5 minutes respectively. We observe that using a short control interval of 1 minute results in significant oscillations in the measured delay. It is due to the fact that the controller reacts ineffectively in response to the measured noise in delay. It is necessary to use a longer control interval so that the controller can allocate servers effectively based on more accurate and consistent measurements of delay.

On the other hand, using a long control interval of 5 minutes reduces the measurement noise, but also reduces the responsiveness of the controller in achieving the end-to-end delay target. In such cases, the control actions of adding or removing virtual servers are performed less frequently. Hence, it takes longer time to bring the end-to-end delay close to the target as shown in Figure 3.42(b).

In the case study, the control interval of 3 minutes is chosen to balance the tradeoff between the neural fuzzy controller’s responsiveness and robustness to noise. The length of the control interval is a very interesting and important issue. There might not exist a control interval that works best for different workload scenarios and different applications.
3.5 Summary

We proposed an efficient server provisioning approach based on an end-to-end resource allocation optimization model. Compared to an existing representative approach, it is able to significantly reduce the number of servers allocated for the end-to-end response time guarantee of multi-tier Internet applications. We further designed a novel self-adaptive neural fuzzy control based server provisioning approach to guarantee the end-to-end delay of requests flowing through multi-tier server clusters. The approach can effectively provide any percentile and the mean based delay guarantee. The major contributions lie in the design and evaluation of a model-independent and self-adaptive control system for dynamic server provisioning. We combine the strength of both machine learning and control theoretic techniques for robust performance assurance in the face of highly dynamic workloads. We have also studied the impact of input scaling factor on controller’s self adaptivity and the impact of the control interval on the system robustness.

Our approach is capable of automatically constructing the controller’s structure and adapting control parameters through fast online learning. Simulation results have demonstrated that the neural fuzzy controller is robust to highly dynamic workloads and changes in delay target. Compared to the rule based fuzzy controller and a classical PI controller, the new approach has shown superior performance in achieving the end-to-end delay assurance, particularly under highly dynamic workloads. Importantly, we have demonstrated the feasibility and excellent performance of the new approach in a testbed implementation of virtualized server cluster. The neural fuzzy controller demonstrated its promise of being a self-adaptive approach for autonomic computing in virtualized datacenters.
Chapter 4

Performance Isolation in Virtualized Datacenters

Server virtualization enables cloud service providers to encapsulate customer applications in VMs and consolidate them on multi-core servers. However, the quality of service experienced by applications hosted in a virtualized server may be significantly impacted by the performance interference between VMs. It is mainly due to the contention of resources such as the last level cache, memory bandwidth, etc, which are shared by VMs co-located on a multi-core processor. Existing techniques that tackle this issue require either hardware architecture-level support or intrusive modification of the guest operating system and virtualization management layer with high implementation overhead. Hence, they have limited applicability in practice.

4.1 NINEPIN: Non-invasive and Energy efficient Performance Isolation

We propose and develop a non-invasive and energy-efficient mechanism that mitigates the performance interference between heterogeneous applications hosted in virtualized servers. As shown in Figure 4.1, NINEPIN interacts with the virtualization management layer of a multi-core server and the co-located VMs at the application layer. It does not require modifications at the underlying systems so that it is easy to use and also more secure for both cloud providers and clients. It applies a powerful fuzzy modeling technique to predict individual application performance as well as energy usage for various allocation of virtualized resources in the presence of complex interference between VMs. A key strength of fuzzy modeling is its ability to capture the inherently non-linear behavior of real world systems over a wide range of operating conditions. Furthermore, with the help of online machine learning, the model is self-adaptive in the face of dynamic vari-
Figure 4.1: NINEPIN: non-invasive performance isolation in virtualized servers.

ations of the workload and environment. NINEPIN utilizes this model and the principles of Model Predictive Control theory to dynamically adjust the VM resources to provide performance isolation among co-located applications in an energy efficient manner.

The key design issues of NINEPIN are as follows:

1. *Non-invasiveness with utility optimization*: An intuitive approach of non-invasive performance isolation among co-located applications is to allocate additional resources to achieve the performance that customers would have realized if they were running in isolation. However, such approaches are inherently utility-agnostic. Integration of non-invasive performance isolation with utility optimization would require highly complex system modeling and computationally expensive control. NINEPIN addresses the challenge by using a novel hierarchical control framework.

2. *Energy efficiency*: A common technique to reducing server energy consumption is to dynamically transition the hardware components from high power states to low-power states. However, it is not applicable in case of virtualized servers since changing the power state of a processor will affect the performance of multiple VMs running different applications. NINEPIN achieves energy efficiency by controlling the CPU usage limits on each VM, based on an accurate energy model. It allows a datacenter administrator to flexibly trade-off energy consumption with the service-level utility of hosted
3. **Robust performance isolation**: The robustness of performance isolation against application heterogeneity and dynamic workload variations requires a self-adaptive approach that responds to the changes in the performance interference relationship and the energy consumption characteristic of the co-located VMs. NINEPIN achieves this goal through a machine learning based online adaptation of the performance interference and energy models of the system, the subsequent re-computation of optimal performance targets and a robust model predictive control based target tracking.

### 4.1.1 NINEPIN Architecture

Figure 5.2.1 presents the architectural overview of the management components used in NINEPIN. The computer system under control is a virtualized server hosting multiple customer applications in VMs that logically abstract the resources provided by the underlying multi-core server. In case of interactive multi-tier applications, each tier of an application is deployed at a virtual machine. The NINEPIN framework forms a control loop that non-invasively mitigates the performance interference between co-located VMs by adjusting their CPU resource allocation (i.e., CPU usage limits) in an energy efficient manner, so that the overall system utility is maximized. The key components in the control loop include a two-level hierarchical controller, a power monitor, a performance monitor for each VM and a resource controller. The two-level control framework integrates utility optimization with control theoretical approach while avoiding highly complex system modeling and computationally expensive control.

#### 4.1.1.1 Power and Performance Monitors

The power monitor measures the average power consumption of the underlying multi-core server for the last control interval.

The performance monitor measures the average performance of the hosted applications in the last control interval. The actual performance metrics may vary for heterogeneous applications running in virtualized environments. Our design does not use any semantic information regarding these performance metrics. It treats the performance values as raw data for modeling and control. Hence, NINEPIN is applicable to any
4.1.1.2 Level-1 Control

At level-1, the utility optimizer calculates the optimal performance targets for each VM in order to maximize the overall system utility and sends the calculated targets to the level-2 controller. The optimization is based on fuzzy MIMO models that capture the performance interference relationship between co-located VMs and the energy consumption property of the underlying server for various CPU resource allocations. These models are constructed offline by applying machine learning techniques on various data collected from the system as described in Section 7.2.2.

It periodically collects the values of power consumption from the power monitor, average performance of running applications from the performance monitor and the CPU usage limits on various VMs from server logs. Then, it calculates the corresponding energy usage due to various applications running in the virtualized server. The total energy usage is a product of the average power consumption and the average completion time of the longest running application.

The measured values are compared with the values of energy usage and performance predicted by the fuzzy MIMO models. If there are significant prediction errors, the fuzzy MIMO models are updated based on the new observations and the optimal performance targets are re-calculated. Such prediction errors can occur
due change in workload.

4.1.1.3 Level-2 Control

At level-2, the model predictive controller computes the CPU usage limits to be enforced on each VM in order to track the optimal performance targets set by the utility optimizer. For this purpose, it uses a linear state-space performance interference model, which is obtained by linearizing the fuzzy MIMO model at each operating point. Linearization reduces the computational complexity of the control problem. It is designed to achieve the performance targets while maintaining system stability in spite of the inevitable uncertainties and disturbances in the system.

The CPU resource allocator is the actuator for this control system. It performs the control actions by enforcing the computed CPU usage limits on the co-located VMs in order to regulate the system towards the optimal targets. Applying CPU usage limits affects a VM’s performance as well as power consumption. It is due to the idle power management of modern processors, which can achieve substantive energy savings when a processor is idle compared to it is active.

4.2 Performance Interference and Energy Usage Modeling

The performance interference and energy usage models are based on fuzzy MIMO modeling technique. In this section, we formulate a fuzzy MIMO model to represent a virtualized multi-core server system hosting multiple applications and discuss a machine learning based model construction technique. MIMO modeling is well-suited to capture the performance interference interactions between co-located VMs. Together with fuzzy logic, it accurately represents the highly complex and nonlinear relationship between various system variables. This is important for achieving modeling accuracy and self-adaptiveness of the system model at the same time. Although the initial fuzzy model is learned for a group of applications, it is adaptive to different workload mixes at run time.
4.2.1 Fuzzy MIMO Model Formulation

We consider a number of applications hosted in a multi-core server as a MIMO system. The inputs to the system are CPU usage limits set for various applications. The outputs of the system are the measured performance of each application and the energy usage of the underlying server. We obtain two separate models for energy usage and performance of the system, respectively. The system is approximated by a collection of MIMO fuzzy models as follows:

\[ y(k + 1) = R(\xi(k), u(k)). \]  

Let \( y(k) \) be the output variable and \( u(k) = [u_1(k), \ldots, u_m(k)]^T \) be the vector of current inputs at sampling interval \( k \). The regression vector \( \xi(k) \) includes current and lagged outputs:

\[ \xi(k) = [y(k), \ldots, y(k - n_y))]^T \]  

where \( n_y \) specifies the number of lagged values of the output variable. Note that a regression vector may also include lagged inputs to achieve even better accuracy of energy usage and performance prediction. \( R \) is a rule-based fuzzy model consisting of \( K \) fuzzy rules. Each fuzzy rule is described as follows:

\[ R_i: \text{If } \xi_1(k) \text{ is } \Omega_{i,1} \text{ and } \ldots \xi_q(k) \text{ is } \Omega_{i,q} \text{ and } u_1(k) \text{ is } \Omega_{i,q+1} \text{ and } \ldots \text{ u}_m(k) \text{ is } \Omega_{i,q+m} \text{ then } \]

\[ y_i(k + 1) = \zeta_i \xi_i(k) + \eta_i u(k) + \phi_i. \]  

Here, \( \Omega_i \) is the antecedent fuzzy set of the \( i^{th} \) rule which describes elements of regression vector \( \xi(k) \) and the current input vector \( u(k) \) using fuzzy values such as ‘large’, ‘small’, etc. \( \zeta_i \) and \( \eta_i \) are vectors containing the consequent parameters and \( \phi_i \) is the offset vector. \( q \) denotes the number of elements in the regression vector \( \xi(k) \). Each fuzzy rule describes a region of the complex non-linear system model using a simple functional relation given by the rule’s consequent part. The model output is calculated as the weighted average of the linear consequents in the individual rules. That is,

\[ y(k + 1) = \frac{\sum_{i=1}^{K} \beta_i (\zeta_i \xi_i(k) + \eta_i u(k) + \phi_i)}{\sum_{i=1}^{K} \beta_i} \]  

where the degree of fulfillment for the \( i^{th} \) rule \( \beta_i \) is the product of the membership degrees of the antecedent variables in that rule. Membership degrees are determined by fuzzy membership functions associated with
the antecedent variables. The model output is expressed in the form of

\[ y(k+1) = \zeta^* \xi_i(k) + \eta^* u(k) + \phi^*. \] (4.5)

The aggregated parameters \( \zeta^* \), \( \eta^* \) and \( \phi^* \) are the weighted sum of vectors \( \zeta_i \), \( \eta_i \) and \( \phi_i \) respectively.

\[
\zeta^* = \frac{\sum_{i=1}^{K} \beta_i \cdot \zeta_i}{\sum_{i=1}^{K} \beta_i}, \\
\eta^* = \frac{\sum_{i=1}^{K} \beta_i \cdot \eta_i}{\sum_{i=1}^{K} \beta_i}, \\
\phi^* = \frac{\sum_{i=1}^{K} \beta_i \cdot \phi_i}{\sum_{i=1}^{K} \beta_i}.
\]

4.2.2 Machine Learning Based Model Construction

We construct initial fuzzy models by applying a subtractive clustering technique on data collected from the system. Each obtained cluster represents a certain operating region of the system, where input-output data values are highly concentrated. The clustering process partitions the input-output space and determines the number of fuzzy rules and the shape of membership functions. Then, we apply an adaptive network based fuzzy inference system (ANFIS) to further tune the fuzzy model parameters. It constructs an artificial neural network to represent a fuzzy model and tunes its parameters using a combination of back-propagation algorithm with a least squares method. This adjustment allows the fuzzy system to learn from the data it is modeling. The data set includes various values of energy usage and performance measured from the system for past resource allocations.

4.2.3 Online Model Adaptation for Robust Performance Isolation

NINEPIN provides robust performance isolation with heterogeneous application support. It addresses the practical issue of hosting compute intensive and interactive applications in the same virtualized server in two steps. First, it performs online adaptation of the performance interference and energy usage models in response to a significant workload variation of interactive applications that are co-located with VMs running compute intensive jobs. Then, it re-computes the optimal performance targets corresponding to the updated
system model. Furthermore, the model predictive controller performs control actions based on the updated performance interference model. Hence, NINEPIN is robust against variations in the workload and heterogeneity of the hosted applications.

The online model adaptation is performed only when a significant error in the prediction of energy usage and performance is detected. This avoids the overhead of frequent adaptation and computationally expensive re-optimization. NINEPIN applies a wRLS (weighted Recursive Least Squares) method to adapt the consequent parameters of the fuzzy MIMO model as new measurements are sampled from the runtime system. It applies exponentially decaying weights on the sampled data so that larger weights are assigned to more recent observations.

For online model adaptation, we express the fuzzy model output in Eq. (5.31) as follows:

$$y(k + 1) = X\theta(k) + e(k)$$ \hspace{1cm} (4.6)

where \(e(k)\) is the error value between actual output of the system (i.e., measured performance) and predicted output of the model. \(\theta = [\theta_1^T \theta_2^T \ldots \theta_p^T]\) is a vector composed of the model parameters. \(X = [w_1 X(k), w_2 X(k), \ldots, w_p X(k)]\) where \(w_i\) is the normalized degree of fulfillment of \(i^{th}\) rule and \(X(k) = [\xi_i^T(k), u(k)]\) is a vector containing current and previous outputs and inputs of the system. The parameter vector \(\theta(k)\) is estimated so that the following cost function is minimized. That is,

$$Cost = \sum_{j=1}^{k} \lambda^{k-j}e^2(j).$$ \hspace{1cm} (4.7)

Here \(\lambda\) is a positive number smaller than one. It is called “forgetting factor” as it gives larger weights on more recent samples in the optimization. This parameter determines in what manner the current prediction error and old errors affect the update of parameter estimation. The parameters of fuzzy model are updated by the wRLS method.

$$\theta(k) = \theta(k - 1) + Q(k)X(k - 1)[y(k) - X(k - 1)\theta(k - 1)].$$ \hspace{1cm} (4.8)

$$Q(k) = \frac{1}{\lambda}[Q(k - 1) - \frac{Q(k - 1)X(k - 1)X^T(k - 1)Q(k - 1)}{\lambda + X^T(k - 1)Q(k - 1)X(k - 1)}].$$ \hspace{1cm} (4.9)

\(Q(k)\) is the updating matrix. The initial value of \(\theta(0)\) is equal to the value obtained in the off-line identification.
4.3 Utility Optimization

The utility optimizer is responsible for finding the optimal service level for each application so that the overall system utility is maximized. It maintains the knowledge about the service-level utility function for each application, the utility function of energy consumption, the performance interference model of co-located VMs and the energy usage model of the underlying multi-core server. The service-level utility function reflects the revenue or penalty related to service-level agreements with customers, and may also incorporate additional considerations such as the value of maintaining the datacenter’s reputation for providing good service. It is of the form $U_i(S)$ for application $i$, where $S$ is the service level achieved in terms of its average performance. The energy utility $U(E)$ represents the costs associated with energy consumption. For a given combination of CPU resource allocations to various co-located applications, the performance interference model specifies the service levels that each application can achieve and the energy usage model specifies the energy consumption of the virtualized server.

The optimization problem is formulated as follows:

$$\text{Maximize} \sum_{i=1}^{N} U_i(S) + \epsilon \ast U(E) \quad (4.10)$$

where $\epsilon$ is a tunable coefficient expressing the relative value of energy efficiency and performance objectives. The utility optimizer first computes the CPU usage limits to be enforced on co-located VMs so that the overall system utility given by Eq. (4.10) is maximized. Then, it determines the optimal service level targets for each application corresponding to the computed CPU usage limits by using the performance interference model. Our optimization algorithm is shown in Algorithm 3.

Various heuristic optimization algorithms such as Simulated Annealing, Genetic Algorithm, Hill Climbing and Particle Swam are well-suited to the optimization problem, due to the complex non-linear relationship between the objective function and the decision variables. Here, the decision variables are the CPU usage limits to be enforced on co-located VMs. In this work, we apply a genetic algorithm based technique that searches the space of various possible CPU usage limits and finds a near-to-optimal solution. It uses the negative of utility optimization objective in Eq. (4.10) as the fitness function since the genetic algorithm is designed to minimize the fitness function. As a result, it maximizes the system utility. We represent a solution
to the optimization problem by a chromosome. It is a string of numbers, coding information about the decision variables. The genetic algorithm generates a new population of candidate solutions and evaluates their fitness values in various iterations. We observe that it is able to converge within 600 iterations or generations.

Algorithm 1 The optimization algorithm.

1: Start with a random initial population where each individual represents a combination of CPU usage limits on co-located VMs.

2: repeat

3: Evaluate each individual solution’s fitness according to the defined fitness function.

4: Select pairs to mate from best-ranked individuals based on their fitness scores.

5: Apply crossover and mutation operations on the selected pairs to generate a new population.

6: until Number-of-generations \( \leq G \)

7: Calculate the optimal performance targets corresponding to the final solution of CPU usage limits, based on the performance interference model.

4.4 Model Predictive Controller

NINEPIN applies the model predictive control principle to regulate the virtualized server system’s dynamic behavior towards the optimal performance targets. The main advantage of using a control theoretical foundation is the ability to achieve the performance targets with better control accuracy and stability in spite of the inevitable uncertainties and disturbances that exist in the system.

4.4.1 Linearized State-Space Model

To apply the model predictive control theory on the virtualized server system, we first linearize the fuzzy MIMO model and represents it as a state-space linear time variant model in the following form:

\[
\begin{align*}
x_{lin}(k+1) &= A(k)x_{lin}(k) + B(k)u(k), \\
y(k) &= C(k)x_{lin}(k).
\end{align*}
\] (4.11)
where \( \zeta \) is as follows: degree of fulfillment of each application and the current CPU usage limits measured from the runtime system. We calculate the aggregated parameters \( \eta \) for the current inputs (i.e. CPU usage limits) chosen for the system and compute the aggregated parameters \( \zeta^*, \eta^* \) and \( \phi^* \). Comparing Eq. (5.32) and Eq. (5.34), the state matrices are computed as follows:

\[
x_{lin}(k + 1) = [\xi^T(k), 1]^T. \tag{4.12}
\]

The matrices \( A(k), B(k) \) and \( C(k) \) are constructed by freezing the parameters of the fuzzy model at a certain operating point \( y(k) \) and \( u(k) \) as follows. The current operating point is determined by the performance values of each application and the current CPU usage limits measured from the runtime system. We calculate the degree of fulfillment \( \beta_i \) for the current inputs (i.e. CPU usage limits) chosen for the system and compute the aggregated parameters \( \zeta^*, \eta^* \) and \( \phi^* \). Comparing Eq. (5.32) and Eq. (5.34), the state matrices are computed as follows:

\[
A = \begin{bmatrix}
\zeta^*_{1,1} & \zeta^*_{1,2} & \cdots & \zeta^*_{1,i} & \phi^*_1 \\
1 & 0 & \cdots & 0 & 0 \\
0 & 1 & \cdots & 0 & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
0 & \vdots & \cdots & \vdots & \vdots \\
\zeta^*_{p,1} & \zeta^*_{p,2} & \cdots & \zeta^*_{p,i} & \phi^*_p \\
0 & 0 & 1 & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
0 & \vdots & \cdots & \vdots & \vdots \\
0 & \cdots & 0 & 0 & 0 & 1
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
\eta^*_{1,1} & \eta^*_{1,2} & \cdots & \eta^*_{1,m} \\
0 & \cdots & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & \cdots & \cdots & 0 \\
\eta^*_{2,1} & \eta^*_{2,2} & \cdots & \eta^*_{2,m} \\
0 & \cdots & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & \cdots & \cdots & 0 \\
\eta^*_{p,1} & \eta^*_{p,2} & \cdots & \eta^*_{p,m} \\
0 & \cdots & \cdots & 0 \\
0 & \cdots & \cdots & 0
\end{bmatrix}
\]

\[
C = \begin{bmatrix}
1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
0 & \cdots & \cdots & 1 & 0
\end{bmatrix}
\]

where \( \zeta^*_{ij} \) is the \( j^{th} \) element of aggregate parameter vectors \( \zeta^* \) for application \( i \). Similarly, \( \eta^*_{ij} \) is the \( j^{th} \) element of aggregate parameter vectors \( \eta^* \) for application \( i \).
4.4.2 MIMO Control Problem

The goal of the model predictive controller is to steer the system into a state of optimum target tracking, while penalizing large changes in the control variables. It minimizes the deviation of application performance from their respective targets. The control decisions are made at every control period $k$ by minimizing the following cost function:

$$V(k) = \sum_{i=1}^{H_p} ||r - y(k + i)||_P^2 + \sum_{j=0}^{H_c-1} ||\Delta u(k + j)||_Q^2.$$  \hspace{1cm} (4.13)

Here, $y(k)$ is a vector containing the performance measure of each application. The controller uses the linearized state-space model to predict each application’s performance over $H_p$ control periods, called the prediction horizon. It computes a sequence of control actions $\Delta u(k), \Delta u(k + 1), ..., \Delta u(k + H_c - 1)$ over $H_c$ control periods, called the control horizon, to keep the predicted performance close to their pre-defined targets $r$. The control action is the change in CPU usage limits imposed on various applications. $P$ and $Q$ are the weighting matrices whose relative magnitude provides a way to tradeoff tracking accuracy for better stability in the control actions.

The control problem is subject to the constraint that the sum of CPU usage limits assigned to all applications must be bounded by the total CPU capacity of the physical server. The constraint is formulated as:

$$\sum_{j=1}^{M} (\Delta u_j(k) + u_j(k)) \leq U_{max}$$  \hspace{1cm} (4.14)

where $M$ is the number of applications hosted in a resource pool and $U_{max}$ is the total CPU capacity of the resource pool.

4.4.3 Transformation to Quadratic Programming Problem

We transform the control formulation to a standard quadratic programming problem, which allows us to design and implement the control algorithm based on an effective quadratic programming method. The MIMO control problem defined by Eq. (5.33) is transformed to a quadratic program:

$$\text{Minimize} \quad \frac{1}{2} \Delta u(k)^T H \Delta u(k) + c^T \Delta u(k)$$  \hspace{1cm} (4.15)

subject to constraint $\Omega \Delta u(k) \leq \omega$. 
The matrices $\Omega$ and $\omega$ are chosen to formulate the constraints on CPU resource usage. Here, $\Delta u(k)$ is a matrix containing the CPU usage limits on each virtual machine over the entire control horizon $H_c$. In the minimization formulation,

$$H = 2(R_u^T P R_u + Q).$$

$$c = 2[R_u^T P^T (R_x A x(k) - r)]^T. \quad (4.16)$$

The matrices $R_u$ and $R_x$ are associated with the performance interference model of the hosted applications.

$$R_u = \begin{bmatrix} C \\ CA \\ \vdots \\ CA^{H_p - 1} \end{bmatrix} \quad R_x = \begin{bmatrix} CB & 0 & \ldots & 0 \\ CAB & CB & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ CA^{H_p - 1}B & CA^{H_p - 1}B & \ldots & CA^{H_p - H_c}B \end{bmatrix}$$

### 4.5 Implementation

#### 4.5.1 The Testbed

We have implemented NINEPIN on a testbed of an HP ProLiant BL460C G6 blade server module and an HP EVA storage area network with 10 Gbps Ethernet and 8 Gbps Fibre/iSCSI dual channels. The blade server is equipped with Intel Xeon E5530 2.4 GHz quad-core processor and 32 GB PC3 memory. Xeon processor incorporates a three level cache hierarchy, where each core has its own L1 (32KB) and L2 (256KB) caches, and there is a large shared 8MB L3 cache. Virtualization of the server is enabled by an enterprise-level virtualization product, VMware ESX 4.1. VMware’s vSphere module controls the CPU usage limits in MHz allocated to the VMs. It also provides an API to support the remote management of VMs. We create a resource pool from the virtualized server to host multiple applications. Each application is hosted inside a VMware virtual machine with one VCPU, 4 GB RAM and 15 GB hard disk space. We assign the CPU affinity of each VM to a particular CPU core. The guest operating system used is Ubuntu Linux version 10.04.

Our testbed utilizes four VMs on the same quad-core processor to host a set of four CPU bound benchmark applications from the SPEC CPU2006 suite. We choose five of the SPEC CPU2006 benchmarks that are identified as being cache sensitive in study [85], and use all possible combinations of four as experimental workload mixes shown in Figure 4.3.
For experiments with heterogeneous application environment, we use SPEC CPU2006 benchmark with the popular RUBiS benchmark [43, 71, 103]. RUBiS is an open-source multi-tier application benchmark. It implements the core functionality of an eBay like auction site: selling, browsing and bidding. The application contains a Java-based client that generates a session-oriented workload. RUBiS sessions have an average duration of 15 minutes and the average think time is five seconds. We use three VMs to host a three-tier RUBiS application and the fourth VM to host one SPEC CPU2006 benchmark application. We instrument RUBiS clients to generate workloads of time-varying intensity.

4.5.2 NINEPIN Components

We implement the components of the NINEPIN framework on a separate machine and issue commands to the virtualized server over the network using VMware vSphere API 4.1.

1. Power Monitor: The average power consumption of the virtualized server is measured at the resource pool level by using a new feature of VMware ESX 4.1. VMware gathers such data through its Intelligent Power Management Interface sensors. The power monitor program uses vSphere API to collect the power measurement data.

2. Performance Monitor: It uses a sensor program provided by RUBiS client for performance monitoring of the interactive application in terms of average end-to-end request response time. For compute intensive jobs, it measures the average job completion time of each VM running the SPEC CPU2006 benchmark application.

3. Performance Interference and Energy Usage Modeling: It applies subtractive clustering and ANFIS
Figure 4.4: Service-level utility of various SPEC CPU2006 applications.

techniques on the data collected from the virtualized server system to construct performance interference and energy usage models. The fuzzy logic toolbox in MATLAB is invoked for this purpose.

4. Hierarchical Controller: It applies a genetic algorithm for system utility optimization and invokes a quadratic programming solver, \textit{quadprog}, in MATLAB to execute the control algorithm described in Section 4.4. We used MATLAB Builder JA to create a Java class from the MATLAB program invoking quadprog. This Java class is integrated into NINEPIN source code. The solution of the control algorithm in terms of VM CPU usage limits is sent to the resource allocator.

5. Resource Allocator: It uses vSphere API to impose CPU usage limits on the VMs. The vSphere module provides an interface to execute a method \textit{ReconfigVMTask} to modify a VM’s CPU usage limit.

4.5.3 NINEPIN Overhead

The computational overhead of NINEPIN is dominated by the quadratic programming problem described in Section 4.4.3, which is solved by applying interior point methods. It has a computational complexity of $O(N)$ Newton iteration, where $N$ is the number of decision variables that need to be computed to solve the given problem. Each Newton iteration requires $O(N^3)$ algebraic operations. NINEPIN runs on a separate machine so that it does not interfere with the performance of the applications hosted in the virtualized server. It is a lightweight solution with a small demand for computational resources.
Table 1: Performance of workload mix-1’s CPU2006 benchmark applications without performance isolation.

<table>
<thead>
<tr>
<th>Application</th>
<th>Completion Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Running alone</td>
</tr>
<tr>
<td>436.cactusADM</td>
<td>2433</td>
</tr>
<tr>
<td>437.leslie3d</td>
<td>1091</td>
</tr>
<tr>
<td>459.GemsFDTD</td>
<td>1190</td>
</tr>
<tr>
<td>470.lbm</td>
<td>825</td>
</tr>
</tbody>
</table>

4.6 Evaluation

For performance evaluation, we consider various service-level utility functions of SPEC CPU2006 applications as shown in Figure 4.4. We chose these utility functions as a case study without any loss of generality. We consider that the utility of energy consumption is given by a linear utility function as follows:

\[ U(E) = \epsilon \times \text{Energy} \]  

where \( \text{Energy} \) is the total energy consumed by virtualized resource pool hosting multiple SPEC CPU2006 applications and \( \epsilon \) is a negative constant, which expresses the relative value of energy and performance objectives. Note that the applicability of NINEPIN in virtualized servers is independent of the chosen utility functions.

We use SPEC CPU2006 suite’s runspec tool to run various benchmarks simultaneously on the co-located VMs. Each benchmark is run on multiple iterations to measure its average performance in terms of the SPECspeed metric. It is amount of time taken to complete a single task. The energy usage is measured in kilojoules (kJ). It is a product of the average power consumption and the average task completion time of the benchmark with the longest running tasks.

4.6.1 Performance Isolation

First, we study the impact of performance interference between co-located applications on their performance. We consider SPEC CPU2006 workload mix 1 that consists of CPU 2006 benchmark applications 436.cac-
Figure 4.5: Performance isolation by default, Q-Clouds and NINEPIN.

tusADM, 437.leslie3d, 459.GemsFDTD and 470.lbm. Table 1 compares the average completion time of the four benchmark applications in the workload mix 1 when each is run on an isolated VM as opposed to when all of them are run simultaneously in co-located VMs. All four benchmark applications exhibit performance degradation in the absence of a performance isolation mechanism. For example, for benchmark application 436.cactusADM, the performance degradation is about 20%. For benchmark application 437.leslie3d, the performance degradation is about 20%. We observe an average performance degradation of 36% in the workload mix 1 due to the fact that VMs running on adjacent CPU cores experience contention of the underlying resources.

Next, we use workload mix 1 to evaluate the performance isolation effectiveness of NINEPIN when the service-level utility and energy utility functions are not available. In this case, NINEPIN aims to mitigate the performance interference effects without optimizing the overall system utility.

We assume that all four VMs require 50% of the CPU resource when there is no performance interference. We measure performance isolation in terms of the normalized performance of the VMs when they are co-located in the same virtualized server. The normalization is performed with respect to the performance shown by the VMs when they run in isolation. Figures 4.5(a) and 4.5(b) show that NINEPIN, compared to Q-Clouds and the default case that does not apply any performance isolation mechanism, is able to achieve much better performance isolation among co-located VMs. Figure 4.5(c) shows the CPU resources allocated to mitigate the performance interference between various hosted applications. The improvement in performance isolation by NINEPIN is due to the use of the fuzzy MIMO model, which captures the performance interference relationship more accurately. Due to the space limitation, we omit the results of other workload mixes and
Table 2: Performance targets for SPEC CPU2006 applications.

<table>
<thead>
<tr>
<th>Target Set</th>
<th>CPU Equivalent Performance %</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>436.cactusADM</td>
</tr>
<tr>
<td>1</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
</tr>
<tr>
<td>3</td>
<td>75</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>70</td>
</tr>
</tbody>
</table>

(a) System utility.  
(b) Energy consumption.  
(c) Variation in optimal target.

Figure 4.6: Optimal performance targets achieved by NINEPIN.

refer to the studies with workload mix 1 as the representative.

4.6.2 Optimal Performance Targeting

We evaluate the merits of utility optimization based performance targeting by NINEPIN. We define a performance target set as a group of performance targets for the applications co-located in a virtualized server. Each performance target is specified as the desired CPU equivalent performance that is the percentage of CPU resource required to achieve a certain performance level [100].

We consider SPEC CPU2006 workload mix 1 that consists of benchmark applications 436.cactusADM, 437.leslie3d, 459.GemsFDTD and 470.lbm. Table 2 gives the performance target sets. Figures 4.6(a) and 4.6(b) compare the system utility and energy consumption associated with the performance target sets. We observe that the performance target set 5, which is computed with NINEPIN, is able to maximize the system utility and minimize the energy consumption. Furthermore, Figure 4.6(c) shows that the optimal performance targets vary with all five different SPEC CPU2006 workload mixes. It is due to the variation in performance interference relationship and the service-level utility functions corresponding to the applications of different workload mixes. Nevertheless, NINEPIN is able assure the optimal performance targets.
**Table 3: Utility and Energy Efficiency.**

<table>
<thead>
<tr>
<th>Workload Mix</th>
<th>System Utility Q-Clouds</th>
<th>System Utility NINEPIN</th>
<th>Energy Consumption (KJ) Q-Clouds</th>
<th>Energy Consumption (KJ) NINEPIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1667</td>
<td>1961</td>
<td>100.8</td>
<td>74.67</td>
</tr>
<tr>
<td>2</td>
<td>1467</td>
<td>1942</td>
<td>115.2</td>
<td>55.72</td>
</tr>
<tr>
<td>3</td>
<td>1500</td>
<td>1955</td>
<td>122.4</td>
<td>82.8</td>
</tr>
<tr>
<td>4</td>
<td>1400</td>
<td>1840</td>
<td>104.4</td>
<td>79.2</td>
</tr>
<tr>
<td>5</td>
<td>1470</td>
<td>1900</td>
<td>118.8</td>
<td>86.4</td>
</tr>
</tbody>
</table>

![Figure 4.7](image.png)

(a) System utility.  
(b) Energy consumption.  
(c) Improvement.

Figure 4.7: System utility and energy efficiency comparison between Q-Clouds and NINEPIN.

### 4.6.3 System Utility and Energy Efficiency

A utility based model provides a practical way to integrate performance assurance and energy efficiency goals of datacenter applications to maximize the profitability of cloud service provider. Table 3 compares the overall system utility and energy efficiency between NINEPIN and Q-Clouds for various workload mixes of SPEC CPU2006 suite. Note that both approaches can mitigate performance interference between co-located applications. However, NINEPIN provides significantly lower energy consumption while improving the system utility.

As shown in Figures 4.7(a) and 4.7(c), NINEPIN is able to achieve better system utility than Q-Clouds for all five workload mixes of SPEC CPU2006 suite. The system utility is a combination of the service-level utility of various co-located applications and the utility of energy consumption. NINEPIN hierarchical control framework maximizes the overall system utility by finding the optimal performance targets based on utility optimization and regulating the system to achieve the reference targets using the model predictive controller. The regulatory action takes place in the form of CPU resource allocations to co-located VMs in the virtualized server. The system utility and energy consumption varies with workload mixes. It is because different combination of applications co-located in a virtualized server manifest different performance interference.
relationships, energy consumption patterns and service-level utility functions. On average, the improvement in the system utility by NINEPIN is 28%.

Figures 4.7(b) and 4.7(c) illustrate the improvement in energy efficiency by NINEPIN for various workload mixes of SPEC CPU2006 suite. NINEPIN reduces the energy consumption of the virtualized server by controlling each VM’s CPU usage limits according to an energy usage model. It is able to tradeoff the utility of meeting performance objectives with energy efficiency. On the other hand, Q-Clouds always aims to achieve a fixed performance target without considering the cost of energy consumption. On average, the improvement in energy efficiency by NINEPIN over Q-Clouds is 32% running the SPEC CPU2006 mixes.

4.6.4 NINEPIN Robustness

We evaluate the robustness of NINEPIN against application heterogeneity and dynamic workload variation. As a case study, we run one interactive three-tier application RUBiS with a dynamic workload and one SPEC CPU2006 benchmark application, 470.lbm, in the same virtualized server. RUBiS application initially faces
Table 4: Improvement in System Utility and Energy Efficiency.

<table>
<thead>
<tr>
<th></th>
<th>Compared with</th>
<th>RUBiS workload</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>500 clients</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000 clients</td>
</tr>
<tr>
<td>System Utility</td>
<td>Default</td>
<td>20%</td>
</tr>
<tr>
<td></td>
<td>Q-Clouds</td>
<td>7.3%</td>
</tr>
<tr>
<td>Energy Efficiency</td>
<td>Default</td>
<td>12%</td>
</tr>
<tr>
<td></td>
<td>Q-Clouds</td>
<td>9.7%</td>
</tr>
</tbody>
</table>

A workload of 500 concurrent users. At the fifth control interval, the workload intensity is doubled.

The prediction accuracy of NINEPIN’s system models in the face of dynamic workload variation has a significant impact on its robustness. Thus, we first measure the accuracy of the fuzzy MIMO models obtained by NINEPIN for performance and energy usage prediction. The accuracy is measured by the normalized root mean square error (NRMSE), a standard metric for deviation. We compare our results with the modeling technique used in Q-Clouds.

Figures 4.8(a), (b) and (c) show that NINEPIN outperforms Q-Clouds in predicting the performance of co-located applications and the energy usage of the underlying server under different workload intensities. The average improvement in the prediction accuracy of performance and energy usage are 26% and 23% respectively. The improvement is more significant when the workload changes from 500 concurrent users to 1000 concurrent users. It is due to the fuzzy MIMO model’s ability to adapt more effectively to the change in workload and capture the inherent non-linearity of the system.

We measure the system utility and energy usage in the face of a dynamic workload shown in Figure 4.9(a). Figure 4.9(b) illustrates the instantaneous system behavior of the virtualized server under the influence of Q-Clouds and NINEPIN mechanisms for performance isolation. We observe that NINEPIN achieves consistently lower energy consumption and improved system utility as compared to Q-Clouds. At the fifth control interval, there is a sharp decline in the system utility for both performance isolation mechanisms. It is due to a sudden change in the performance interference relationship between heterogeneous applications, which is caused by the workload variation. Furthermore due to increase in the workload intensity, the energy consumption by the underlying server also increases. Indeed, performance interference effects are impacted by the workload intensity as well as characteristics of co-located applications. Note that the performance improvement by NINEPIN is more significant after the fifth control interval. It is due to its ability to re-compute
and assure the optimal operating conditions of the system in response to the changing performance interference relationship between heterogeneous applications. Figure 4.9(c) summarizes the energy consumption improvement by NINPIN.

Table 4 shows the improvement in system utility and energy efficiency by NINEPIN for different RUBiS workloads, compared to Q-Clouds and the default case that does not apply any performance isolation mechanism. In the two scenarios, NINEPIN outperforms Q-Clouds in average energy efficiency and average system utility by 16% and 72%, respectively.

4.7 Summary

Performance isolation among heterogeneous customer applications is an important but very challenging problem in a virtualized datacenter. NINEPIN provides a desirable non-invasive performance isolation mechanism for a datacenter hosting third-party customer applications and using virtualization software from third-party vendors. As demonstrated by modeling, analysis and experimental results based on the testbed implementation, its main contributions are robust performance isolation of heterogeneous applications, energy efficiency and overall system utility optimization. It increases datacenter utility by aligning performance isolation goals with a datacenter’s economic optimization objective. The main technical novelty of NINEPIN is due to the proposed and developed hierarchical control framework that integrates the strengths of machine learning based system modeling, utility based performance targeting and a model predictive control based target tracking and optimization.
Chapter 5

Middleware for Power and Performance Control on Virtualized Servers

Hardware throttling is too rigid for power control in virtualized environments because reducing CPU frequency of a server inevitably affects the performance of all VMs running on that server. Unlike previous approaches, PERFUME controls the power consumption of a virtual resource pool while assuring the performance of multi-tier applications hosted in it. A resource pool aggregates the CPU resources provided by underlying server clusters.

We regulate the power consumption by applying CPU usage limits on VMs hosted on a cluster of blade servers. It constrains the utilization of underlying physical processors thereby regulates power consumption. It is feasible due the idle power management of modern processors, which incorporate sleep states (C-states) to achieve substantive power savings when a processor is idle [82].

![Figure 5.1: PERFUME system overview.](image)

Figure 5.1: PERFUME system overview.
5.1 PERFUME: Coordinated Power and Performance Control on Virtualized Server Clusters

It is important but challenging to assure the performance of multi-tier Internet applications with the power consumption cap of virtualized server clusters mainly due to system complexity of shared infrastructure and dynamic and bursty nature of workloads. We propose and develop PERFUME, a system that simultaneously provides explicit guarantee on the power consumption of underlying server clusters and the performance of multi-tier applications hosted on them. As shown in Figure 5.1, various applications are hosted on a virtualized server cluster according to an interference-aware application placement policy. Such a placement policy decides which applications should be co-located so that the performance interference between applications can be minimized [92]. PERFUME provides the flexibility to the system administrator in setting the power and performance targets, and specifying the trade-offs between the cost of power consumption and the business value of providing performance assurance to various applications.

PERFUME’s core is a fuzzy MIMO (FUMI) controller that minimizes the deviation of power and performance from their respective targets while assuring control accuracy and system stability. FUMI control is capable of dealing with the complexity of multi-tier applications in a shared virtualized infrastructure and the inherent non-linearities that exist in a real Web system. It is due to its integration of fuzzy modeling logic, MIMO control and artificial neural network. The control action is taken by adjusting the CPU usage limits among individual tiers of multiple applications in a coordinated manner. Furthermore, it provides service differentiation by prioritizing CPU allocations among different applications while avoiding power budget violations.

PERFUME provides performance guarantee for throughput and percentile-based response time in the face of highly dynamic and bursty workloads. Its novel FUMI control accurately captures the strong nonlinearity of percentile-based performance metric such as the $95_{th}$-percentile response time by applying fuzzy models. It also predicts the power consumption of underlying server clusters for various CPU usage limits in hosted applications. PERFUME is self-adaptive to highly dynamic workloads due to its online learning capability. It automatically learns the fuzzy model parameters at run time using a weighted recursive least-squares (wRLS)
Furthermore, PERFUME addresses an important challenge of pro-actively avoiding violations of power and performance targets in anticipation of future workload changes. In spite of FUMI control’s ability to adapt itself in the face of workload variations, target violations may occur to some extent. Such violations are even more significant in case of continuously changing workloads as seen in real-world Web traces. It is due to the purely reactive approach of updating the power and performance models in response to the measured modeling errors. Since it takes a few control intervals to accurately update the system model, the control actions may lag behind continuous variations in the workload. We enhance the proposed FUMI control by integrating a workload prediction component. The integration involves workload-aware MIMO fuzzy modeling of the virtualized server system and the design of proactive FUMI control based on this model. Our proactive FUMI control is able to make effective control decisions in anticipation of future workload changes. We apply a standard Kalman filtering technique [58] to predict workload variations.

5.1.1 PERFUME System architecture and Design

Figure 5.2 illustrates the system architecture of PERFUME. The system under control is a virtualized server cluster hosting multiple multi-tier applications. Each tier of an application is deployed at a VM created from a resource pool. The power monitor periodically measures the average power consumption of the server cluster and sends the value to the FUMI control. The performance monitor periodically measures the throughput as
as well as the percentile-based response time of each multi-tier application and sends the performance values to the controller. FUMI control determines the CPU usage limits on the Web, application and database tiers of multiple applications to regulate per-application performance and the total power consumption of the server cluster. The resource allocator actuates the control action to limit the CPU usage of each VM by using the virtualization management module.

5.1.2 Modeling of Coordinated Power and Performance control

We apply fuzzy modeling to predict the performance of multi-tier applications for various CPU usage limits imposed on the VMs. Both throughput and the percentile-based response time are used as performance metrics. Fuzzy modeling also estimates the complex relationship between the power consumption of the resource pool and the CPU usage limits imposed on various tiers of the applications. A key strength of fuzzy model is its ability to represent highly complex nonlinear systems by a combination of inter-linked subsystems with simple functional dependencies. A simple linear model is not sufficient in this case due to the complex inter-tier dependencies and the underlying complexity of virtualized server infrastructure.

5.1.2.1 The Fuzzy Model

We consider a number of multi-tier applications hosted in a virtual resource pool as a MIMO system. The inputs to the system are CPU usage limits set at various tiers of the applications. The outputs of the system are the measured performance of each application and the average power consumption of the shared resource pool. We obtain two separate models for power and performance of the system, respectively. The system is
approximated by a collection of MIMO fuzzy models as follows:

\[ y(k + 1) = R(\xi(k), u(k)). \]  

(5.1)

Let \( y(k) \) be the output variable and \( u(k) = [u_1(k), \ldots, u_m(k)]^T \) be the vector of current inputs at sampling interval \( k \). The regression vector \( \xi(k) \) includes current and lagged outputs:

\[ \xi(k) = [y(k), \ldots, y(k - n_y)]^T \]  

(5.2)

where \( n_y \) specifies the number of lagged values of the output variable. A regression vector may also include lagged inputs for better accuracy of power and performance prediction. \( R \) is a fuzzy model consisting of a set of fuzzy rules. Each fuzzy rule is described as follows:

- \( R_i: \) If \( \xi_1(k) \) is \( \Omega_{i,1} \) and \( \ldots \) \( \xi_{\varrho}(k) \) is \( \Omega_{i,\varrho} \) and \( u_1(k) \) is \( \Omega_{i,1+1} \) and \( \ldots \) \( u_m(k) \) is \( \Omega_{i,1+m} \) then

\[ y_i(k + 1) = \zeta_i \xi(k) + \eta_i u(k) + \phi_i. \]  

(5.3)

Here, \( \Omega_i \) is a set of fuzzy values, which describe the elements of regression vector \( \xi(k) \) and the current input vector \( u(k) \) for the fuzzy rule, \( R_i \). The numeric values of \( \xi(k) \) and \( u(k) \) are mapped to fuzzy values by using the corresponding fuzzy membership functions. For example, the fuzzy membership function of \( \Omega_{i,1} \) determines the degree to which it can accurately describe \( \xi_1(k) \). \( \varrho \) denotes the number of elements in the regression vector \( \xi(k) \). \( y_i(k + 1) \) is the estimated model output according to the fuzzy rule \( R_i \). \( \zeta_i \) and \( \eta_i \) are vectors containing the consequent parameters and \( \phi_i \) is the offset vector.

Note that the fuzzy membership functions may overlap with each other. As a result, multiple fuzzy rules can be triggered by a given set of input values. Each fuzzy rule describes a region of the complex non-linear system model using a simple functional relation given by the rule’s consequent part. The contribution of each rule to the model output is determined by its firing strength, \( \beta_i \). It is the product of the membership degrees of the antecedent variables in that rule. The final model output is calculated as the weighted average of the linear consequents in the individual rules. That is,

\[ y(k + 1) = \frac{\sum_{i=1}^{K} \beta_i (\zeta_i \xi(k) + \eta_i u(k) + \phi_i)}{\sum_{i=1}^{K} \beta_i} \]  

(5.4)

where \( K \) is the total number of fuzzy rules.
We conduct a case study to demonstrate the accuracy of our MIMO fuzzy models in predicting the performance of a RUBiS application and the power consumption of the underlying virtualized server cluster. The data for modeling is collected by randomly allocating various CPU usage limits on the Web, application and database tiers of the RUBiS application, which faces a browsing workload mix of 1000 concurrent users. We construct an initial fuzzy model by applying subtractive clustering technique [21] on data collected from the system. Each obtained cluster represents a certain operating region of the system, where input-output data values are highly concentrated. Clustering partitions the input-output space to determine the number of fuzzy rules and the shape of membership functions.

In this case study, we obtain four clusters in a five dimensional space. The first four dimensions correspond to the three input variables, \([u_1(k), u_2(k), u_3(k)]\), and one regression vector element \(\xi_1(k)\). The fifth dimension corresponds to the output variable, \(y(k)\), which is expressed as a linear function of the input variables. Each cluster center describes a fuzzy rule in which the fuzzy values \(\Omega_i\) corresponding to \(u(k)\) and \(\xi(k)\) are represented by gaussian membership functions. The first four dimensions of the cluster center determine the mean of the gaussian functions. The function variance is determined by a tunable parameter in the subtractive clustering technique. For example, the four cluster centers in our performance model are \([0.46, 0.61, 0.53, 0.6], [0.97, 0.4, 0.63, 0.81], [0.91, 0.87, 0.49, 0.94], \) and \([0.55, 0.65, 0.93, 0.96]\). These are normalized values with respect to the maximum observed values of CPU resource usage limits imposed at the three tiers of the RUBiS application and the average throughput in the previous sampling interval. We apply an adaptive network based fuzzy inference system [53] to further tune the fuzzy model parameters in Eq. (5.30).
Figures 5.3(a), 5.3(b) and 5.3(c) show that our fuzzy models can accurately predict the application throughput, the 95th-percentile response time and the average power consumption for various CPU allocations. The accuracy is measured by the normalized root mean square error (NRMSE), a standard metric for deviation. The case study show that the checking and the predicted data are very close, with the NRMSE 12.5%, 17.6% and 15.2% in the three scenarios respectively. We use different data sets for training and validating the system models.

5.1.2.2 On-line Adaptation of the Fuzzy Model

Internet workloads to a datacenter vary dynamically in arrival rates as well as characteristics [113]. This results in significantly varying resource demands at multiple tiers of Internet applications. A static system model can not provide sufficient prediction accuracy of power and performance for all possible variations in the workload. Hence, the system models need to adapt on-line in the face of dynamic workloads. We apply a wRLS method to adapt the consequent parameters of the fuzzy model obtained. The technique continuously samples new measurements from the runtime system. It updates the model parameters in response to the errors made by the existing fuzzy models in predicting the performance and power consumption. The recursive nature of the wRLS method makes the time taken for this computation negligible for a control interval that is more than 10 seconds. It applies exponentially decaying weights on the sampled data so that higher weights are assigned to more recent observations.

We express the fuzzy model output in Eq. (5.31) as follow:

\[ y(k + 1) = X\theta(k) + e(k) \]  (5.5)

where \( e(k) \) is the error value between actual output of the system (i.e., measured performance or power) and predicted output of the model. \( \theta = [\theta_1^T \theta_2^T .. \theta_p^T] \) is a vector composed of the model parameters. \( X = [w_1 X(k), w_2 X(k), .., w_p X(k)] \) where \( w_i \) is the normalized degree of fulfillment or firing strength of \( i^{th} \) rule and \( X(k) = [\xi^T(k), u(k)] \) is a vector containing current and previous outputs and inputs of the system. The parameter vector \( \theta(k) \) is estimated so that the following cost function is minimized. That is,

\[ Cost = \sum_{j=1}^{k} \gamma^{k-j} e^2(j). \]  (5.6)

Here \( \gamma \) is a positive number less than one. It is called “forgetting factor” as it gives higher weights on more
recent samples in the optimization. It determines in what manner the current prediction error and old errors affect the update of parameter estimation. The parameters of fuzzy model are updated according to the wRLS method as follows:

\[ \theta(k) = \theta(k-1) + Q(k)X(k-1)[y(k) - X(k-1)\theta(k-1)]. \]  

(5.7)

\[ Q(k) = \frac{1}{\gamma}[Q(k-1) - \frac{Q(k-1)X(k-1)X^T(k-1)Q(k-1)}{\gamma + X^T(k-1)Q(k-1)X(k-1)}]. \]  

(5.8)

Here \( Q(k) \) is the updating matrix. The initial value of \( \theta(0) \) is equal to the value obtained in the off-line identification. And, the initial value of \( Q(0) \) is equal to \( (X^T X)^{-1} \).

To evaluate the self-adaptiveness of our fuzzy model, we measure its power and performance prediction accuracy when a RUBiS workload is changed from browsing mix of 1000 concurrent users to bidding mix of 500 concurrent users and vice versa. Our results are compared with a popular and recently used technique for modeling Internet systems, ARMA [43, 103]. As shown in Figures 5.4(a) and 5.4(b), our fuzzy model out-performs ARMA model in predicting performance of a multi-tier application for both stationary and dynamic workloads. On average, the improvement in performance prediction accuracy for the throughput and 95th-percentile end-to-end response time are 35% and 43%, respectively. The improvement in power prediction accuracy is shown in Figure 5.4(c).

Compared to ARMA model, our fuzzy models are more accurate in capturing the inherently non-linear relationship between resource allocation and performance or power in a virtualized server system.

### 5.1.2.3 Integration of workload-aware fuzzy modeling for proactive FUMI Control

Although effective, the online adaptation of fuzzy model takes a few control intervals to capture the changing system behavior. As a result, the control performance may be degraded if the workload is continuously changing. We address this challenge by the integration of workload-aware fuzzy modeling to achieve proactive control in anticipation of future workload changes. The novelty of proactive FUMI control lies in its ability to consider the impact of future workload changes as well as current control actions while solving the MIMO control problem.

We incorporate the time varying workload intensity as a measured disturbance in the system model. The system is now approximated by a collection of MIMO fuzzy models as follows:

\[ y(k+1) = R(\xi(k), \lambda(k), u(k)). \]  

(5.9)
where the workload intensity at the sampling interval \( k \) is denoted by \( \lambda(k) \). Each fuzzy rule in the model is described as follows:

- \( R_i: \) If \( \xi_1(k) \) is \( \Omega_{i,1} \) and .. \( \xi_\varphi(k) \) is \( \Omega_{i,\varphi} \) and \( \lambda(k) \) is \( \Omega_{i,\varphi+1} \) and \( u_1(k) \) is \( \Omega_{i,\varphi+2} \) and .. \( u_m(k) \) is \( \Omega_{i,\varphi+m+1} \) then

\[
y_i(k+1) = \zeta_i \xi(k) + \theta_i \lambda(k) + \eta_i u(k) + \phi_i.
\]

(5.10)

Here, a set of fuzzy values denoted by \( \Omega_i \) describes \( \lambda(k) \) in addition to other model parameters. The model output is calculated as:

\[
y(k+1) = \frac{\sum_{i=1}^{K} \beta_i (\zeta_i \xi(k) + \theta_i \lambda(k) + \eta_i u(k) + \phi_i)}{\sum_{i=1}^{K} \beta_i}
\]

(5.11)

It can also be expressed in the form of

\[
y(k+1) = \zeta^* \xi(k) + \theta^* \lambda(k) + \eta^* u(k) + \phi^*.
\]

(5.12)

The aggregated parameters \( \zeta^*, \theta^*, \eta^* \) and \( \phi^* \) are the weighted sum of vectors \( \zeta_i, \theta_i, \eta_i \) and \( \phi_i \) respectively. They are applied to obtain a state-space system model and transform the complex control problem into a computationally efficient quadratic programming problem, which is described in Section 5.1.3.2.

We construct a workload-aware fuzzy MIMO model by applying subtractive clustering technique and adaptive network based fuzzy inference system on the data collected from the virtualized server system hosting multi-tier applications. For data collection, we measure the power consumption and performance achieved by the system for various combinations of CPU allocations and workload intensities. The power and performance models, which are learnt from the training data, have the ability to generalize the expected system behavior for previously unseen workloads and CPU allocations. Our workload-aware fuzzy model consists of six fuzzy rules.

At run time, we apply the wRLS method to update the model parameters if the system behaves differently than expected by the fuzzy model. Such a situation could arise due to change in the workload characteristics, which is not captured by the model.
5.1.3 FUMI Control Design

We apply the model predictive control principle and fuzzy modeling to design the FUMI control. FUMI control is well suited for power and performance control in virtualized server clusters due to its capability to solve constrained MIMO control problems of complex non-linear systems. It determines control actions by optimizing a cost function, which expresses the control objectives and constraints over a time interval. We formulate the power and performance assurance of virtualized multi-tier applications as a predictive control problem. Then, we present detailed steps to transform the control formulation to a standard quadratic programming problem, which allows us to design and implement the control algorithm based on an effective quadratic programming method.

5.1.3.1 FUMI Control Problem Formulation

FUMI control aims to minimize the deviation of power consumption and performance of multi-tier applications from their respective targets. It decides the control actions at every control period $k$ by minimizing the cost function:

$$
V(k) = \sum_{i=1}^{H_p} ||r_1 - y_1(k+i)||_P^2 + \sum_{i=1}^{H_p} ||r_2 - y_2(k+i)||_Q^2 + \sum_{j=0}^{H_c-1} ||\Delta u(k+j)||_R^2 \tag{5.13}
$$

Here, $y_1(k)$ is the power consumption of the resource pool. $y_2(k)$ is a vector containing the percentile-based end-to-end response time or the throughput of each application. The controller predicts both power and performance over $H_p$ control periods, called the prediction horizon. It computes a sequence of control actions $\Delta u(k), \Delta u(k+1), \ldots, \Delta u(k + H_c - 1)$ over $H_c$ control periods, called the control horizon, to keep the predicted power and performance close to their pre-defined targets $r_1$ and $r_2$ respectively. The control action is the change in CPU usage limits imposed on various tiers of the multi-tier applications. $P$ and $Q$ are the tracking error weights that determine the trade-off between power and performance. The tracking error weights also facilitate service differentiation between different applications sharing the resource pool. The third term represents the control penalty, weighted by $R$. It penalizes big changes in control action and contributes towards system stability.
A system administrator can select the power and performance targets and their respective weighting parameters by using an independent optimization framework, or by following best practices, based on the business value of providing various levels of performance and the cost of power consumption. A higher preference weight can be set for power consumption control, when the power budget needs to be enforced. However, temporary violations of power budgets are allowable, as long as they are bounded. Thermal failover happens only when the power budget is violated long enough to create enough heat that increases the temperature beyond normal operational ranges. Hence, we do not treat power budget as a hard constraint in our problem formulation.

The control problem is subject to the constraint that the sum of CPU usage limits assigned to all multi-tier applications must be bounded by the total CPU capacity of the resource pool. The constraint is formulated as:

$$\sum_{m=1}^{M} (\Delta u_m(k) + u_m(k)) \leq U_{max}$$

(5.14)

where $M$ is the number of applications hosted and $U_{max}$ is the total CPU capacity of the resource pool.

The use of resource pool enables resource management at the server cluster level, independently of the actual hosts that contribute to the resources. Hence, we do not consider the CPU capacity constraint of individual physical server.

### 5.1.3.2 Transformation to Quadratic Programming

We transform the non-convex, time-consuming optimization involved in the MIMO control problem into a standard quadratic programming problem, which can be solved efficiently at run time. We express the objective of FUMI control, defined by Eq. (5.33), as a quadratic program:

$$\text{Minimize} \quad \frac{1}{2} \Delta u(k)^T H \Delta u(k) + c^T \Delta u(k)$$

subject to constraint $\Omega \Delta u(k) \leq \omega$.

The matrices $\Omega$ and $\omega$ are chosen to formulate the constraints on CPU resource usage as described in Eq. (5.14). Here, $\Delta u(k)$ is a matrix containing the CPU usage limits on each VM over the entire control horizon $H_c$.

For this transformation, first we linearize the fuzzy model at the current operating point and represent it
as a state-space linear time variant model in the following form:

\[
    x_{lin}(k + 1) = A(k)x_{lin}(k) + B_u(k)u(k) + B_\lambda(k)\lambda(k),
\]

\[
    y_{lin}(k) = C(k)x_{lin}(k).
\]

(5.16)

The vector for the state-space description is defined as

\[
    x_{lin}(k + 1) = [\xi^T(k), 1]^T.
\]

(5.17)

The matrices \(A(k), B_u(k), B_\lambda(k)\) and \(C(k)\) are constructed by freezing the parameters of the fuzzy model at a certain operating point \(y(k)\) and \(u(k)\). Comparing Eq. (5.12) and Eq. (5.16), the state matrices are computed as follows:

\[
    A = \begin{bmatrix}
        \xi_{1,1} & \xi_{1,2} & \cdots & \xi_{1,o} & \phi_1^i \\
        1 & 0 & \cdots & 0 & 0 \\
        0 & 1 & \cdots & 0 & 0 \\
        \vdots & \vdots & \ddots & \vdots & \vdots \\
        0 & \cdots & \cdots & 0 & 0 \\
    \end{bmatrix}
\]

\[
    B_u = \begin{bmatrix}
        \eta_{1,1} & \eta_{1,2} & \cdots & \eta_{1,m} & \theta_1^i \\
        0 & \cdots & \cdots & 0 & 0 \\
        \vdots & \vdots & \ddots & \vdots & \vdots \\
        \eta_{2,1} & \eta_{2,2} & \cdots & \eta_{2,m} & \theta_2^i \\
        0 & \cdots & \cdots & 0 & 0 \\
        \vdots & \vdots & \ddots & \vdots & \vdots \\
        \eta_{p,1} & \eta_{p,2} & \cdots & \eta_{p,m} & \theta_p^i \\
        0 & \cdots & \cdots & 0 & 0 \\
        0 & \cdots & \cdots & 0 & 0 \\
    \end{bmatrix}
\]

\[
    B_\lambda = \begin{bmatrix}
        0 \\
        \vdots \\
        0 \\
    \end{bmatrix}
\]

\[
    C = \begin{bmatrix}
        1 & 0 & \cdots & \cdots & 0 \\
        \vdots & \vdots & \ddots & \vdots & \vdots \\
        0 & \cdots & \cdots & 1 & 0 \\
    \end{bmatrix}
\]

where \(\xi_{ij}^*\) is the \(j^{th}\) element of aggregate parameter vectors \(\xi^*\) for application \(i\). \(\eta_{ij}^*\) is the \(j^{th}\) element of aggregate parameter vectors \(\eta^*\) for application \(i\). \(\theta_i^*\) is the aggregate parameter \(\theta^*\) for application \(i\).
To ensure offset-free reference tracking, the optimization problem is defined with respect to the increment in the control signal, $\Delta u(k)$, rather than the control signal $u(k)$. The state-space description is extended correspondingly as follows:

$$
\begin{bmatrix}
x_{lin}(k+1) \\
u(k)
\end{bmatrix} =
\begin{bmatrix}
A(k) & B_\Delta(k) \\
0 & I
\end{bmatrix}
\begin{bmatrix}
x_{lin}(k) \\
u(k-1)
\end{bmatrix} +
\begin{bmatrix}
B_\Delta(k) \\
I
\end{bmatrix}
\Delta u(k) +
\begin{bmatrix}
B_\lambda(k) \\
0
\end{bmatrix}
\lambda(k)
$$

$$
y_{lin} =
\begin{bmatrix}
C(k) & 0
\end{bmatrix}
\begin{bmatrix}
x_{lin}(k) \\
u(k-1)
\end{bmatrix}
$$

$$
\uparrow
$$

$$
X(k+1) = \bar{A}X(k) + \bar{B}_\Delta \Delta u(k) + \bar{B}_\lambda \lambda(k).
$$

$$
Y(k) = \bar{C}X(k).
$$

We obtain the state-space description, shown in Eq. (5.18), corresponding to both power and performance models. Henceforth, we use the notations $\bar{A}_1, \bar{B}_{1u}, \bar{B}_{1\lambda}, \bar{C}_1, X_1(k), Y_1(k)$ and $\bar{A}_2, \bar{B}_{2u}, \bar{B}_{2\lambda}, \bar{C}_2, X_2(k), Y_2(k)$ for the state matrices that describe the performance model and the power model respectively.

Assuming that at time instant $k$, the state vector, the future control sequence and the future workload are known, the future process outputs can be predicted through successive substitution. The complete output sequence over the prediction horizon $H_p$ is given by the following:

$$
\begin{bmatrix}
Y_1(k+1) \\
Y_1(k+2) \\
\vdots \\
Y_1(k+H_p)
\end{bmatrix} = R_{tu} \bar{A}_1 X_1(k) + R_{tu} \Delta u(k) +
\begin{bmatrix}
\Delta u(k+1) \\
\vdots \\
\Delta u(k+H_w-1)
\end{bmatrix}
$$

$$
+ R_{t\lambda} \begin{bmatrix}
\lambda(k) \\
\lambda(k+1) \\
\vdots \\
\lambda(k+H_p)
\end{bmatrix}
$$
where

\[
R_{tu} = \begin{bmatrix}
\mathcal{C}_i & \mathcal{A}_i & \\
\mathcal{C}_i & \mathcal{A}_i & \\
\vdots & \vdots & \\
\mathcal{C}_i & \mathcal{A}_i^{H_p-1}
\end{bmatrix}
\]

\[
R_{i\lambda} = \begin{bmatrix}
\mathcal{C}_i B_i & 0 & \ldots & 0 \\
\mathcal{C}_i A_i B_{iu} & \mathcal{C}_i B_{iu} & \ldots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
\mathcal{C}_i A_i^{H_p-1} B_{iu} & \mathcal{C}_i A_i^{H_p-2} B_{iu} & \ldots & \mathcal{C}_i A_i^{H_p-H_p} B_{iu}
\end{bmatrix}
\]

Combining the complete output sequence over the prediction horizon with the FUMI control objective, we get a quadratic programming problem defined in Eq. (5.15) where,

\[
H = 2(\mathcal{R}_{1u}^T P \mathcal{R}_{1u} + \mathcal{R}_{2u}^T Q \mathcal{R}_{2u} + R).
\]

\[
c = 2[\mathcal{R}_{1u}^T P^T (R_{iu} A_1 X_1(k) - r1 + R_{1\lambda} \lambda)]
+ R_{2u}^T Q^T (R_{2u} A_2 X_2(k) - r2 + R_{2\lambda} \lambda)]^T. \tag{5.19}
\]

where

\[
\lambda = \begin{bmatrix}
\lambda(k) \\
\lambda(k+1) \\
\vdots \\
\lambda(k+H_p)
\end{bmatrix}
\]

The integration of workload intensity \( \lambda \) in the FUMI control problem enables proactive control in the face of dynamic workload variations. The future values of workload intensity \( \lambda(k) \) over the prediction horizon \( H_p \) are predicted by applying Kalman filtering technique.

Note that we use the workload-aware fuzzy model for deriving a proactive FUMI control solution, assuming that workload predictions are sufficiently accurate. When the workload variations are abrupt and unpredictable, FUMI control uses the fuzzy model described in Section 5.1.2.1. In this case, we obtain the following:

\[
H = 2(\mathcal{R}_{1u}^T P \mathcal{R}_{1u} + \mathcal{R}_{2u}^T Q \mathcal{R}_{2u} + R).
\]

\[
c = 2[\mathcal{R}_{1u}^T P^T (R_{iu} A_1 X_1(k) - r1) + \mathcal{R}_{2u}^T Q^T (R_{2u} A_2 X_2(k) - r2)]^T. \tag{5.20}
\]
The matrices $R_{1x}, R_{1u}$ are associated with the performance models of hosted applications and matrices $R_{2x}, R_{2u}$ are associated with the power model of the resource pool.

$$R_{1x} = \begin{bmatrix}
\bar{C}_i \\
\bar{C}_i \bar{A}_i \\
\vdots \\
\bar{C}_i \bar{A}_i^{H_p-1}
\end{bmatrix}$$

$$R_{1u} = \begin{bmatrix}
\bar{C}_i \bar{B}_i & 0 & \ldots & 0 \\
\bar{C}_i \bar{A}_i \bar{B}_i & \bar{C}_i \bar{B}_i & \ldots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
\bar{C}_i \bar{A}_i^{H_p-1} \bar{B}_i & \bar{C}_i \bar{A}_i^{H_p-2} \bar{B}_i & \ldots & \bar{C}_i \bar{A}_i^{H_p-H_c} \bar{B}_i
\end{bmatrix}$$

### 5.1.3.3 FUMI Control Interface

Figure 5.5 shows the interface between the FUMI online learning and MIMO control components. The continuous chain of interactions between various components of the FUMI control system is described by the Algorithm 3.

### 5.1.4 Implementation

#### 5.1.4.1 The Testbed

We have implemented PERFUME on a testbed consisting of two HP ProLiant BL460C G6 blade server modules and a HP EVA storage area network with 10 Gbps Ethernet and 8 Gbps Fibre/iSCSI dual channels. Each blade server is equipped with Intel Xeon E5530 2.4 GHz quad-core processor and 32 GB PC3 memory. Virtualization of the cluster is enabled by VMware ESX 4.1. VMware’s vSphere module provides an API to support remote management of VMs. We create a resource pool from the virtualized server cluster to host multi-tier applications. An important feature of VMware resource pool is that the VMs do not have a static mapping with the physical servers. VMware’s Distributed Resource Scheduling mechanism dynamically
Algorithm 2 The Control Algorithm.

1: Start with MIMO fuzzy models that are constructed from offline data collected from the system.

2: loop

3: Linearize the fuzzy models at the current operating state and obtain the state-space matrices: $\bar{A}_1, \bar{B}_{1u}, \bar{B}_{1\lambda}, \bar{C}_1, \bar{A}_2, \bar{B}_{2u}, \bar{B}_{2\lambda},$ and $\bar{C}_2$.

4: Predict the workload intensity of the hosted applications over the prediction horizon, $H_p$, if workload prediction capability is available.

5: Solve the quadratic programming problem described in Section 5.1.3.2 to minimize the deviation of power consumption and performance of multi-tier applications from their respective targets, denoted by $\text{ref}$. Obtain a sequence of CPU resource adjustments, $u(k)$, over the control horizon, $H_c$.

6: Apply the first control action from the computed sequence of actions, in terms of CPU resource adjustments on the Web, Application and database tiers of the hosted applications.

7: Adapt the fuzzy models using the wRLS method, in response to any modeling errors observed.

8: end loop

changes the mapping for load balancing. As shown in Figure 5.2, each tier of an application is hosted inside a VM with 2 VCPUs, 4 GB RAM and 15 GB disk space. The guest operating system used is Ubuntu Linux 10.04.

As many related studies [43, 103, 123, 135], our work uses an open-source multi-tier application benchmark RUBiS in the testbed. RUBiS implements the core functionality of an eBay like auction site: selling, browsing and bidding. It has nine tables in the database and defines 26 interactions that can be accessed from the clients’ Web browsers. The application contains a Java-based client that generates a session-oriented workload. RUBiS sessions have an average duration of 15 minutes and the average think time is five seconds. It defines two workload mixes: a browsing mix made up of only read-only interactions and a bidding mix that includes 15% read-write interactions. We configure the RUBiS clients to generate workloads of different mixes as well as time-varying intensity.
5.1.4.2 PERFUME Components

1. Power Monitor: The average power consumption can be measured at the resource pool level or at the VM level by using a feature of VMware ESX 4.1. VMware gathers such data through its Intelligent Power Management Interface sensors. The power monitor program uses vSphere API to collect the power measurement data.

2. Performance Monitor: It uses a sensor program provided by RUBiS client for performance monitoring. We modify the sensor to measure the client-perceived percentile-based response time and throughput over a period of time. The number of requests finished during a control interval is the throughput.

3. FUMI Controller: It determines the control actions at every interval of 30 seconds. This control interval is chosen by considering the trade-off between noise in the sensor measurements and faster response of the controller. It invokes a quadratic programming solver, quadprog, in MATLAB to execute the control algorithm described in Section 5.1.3. The solution of the control algorithm in terms of VM CPU usage limits is sent to the resource allocator.

4. Resource Allocator: It uses vSphere API to impose CPU usage limits on the VMs. The vSphere module provides an interface to execute a method ReconfigVM_Task for this purpose.

In our testbed, the overhead of applying the wRLS technique on the fuzzy models and executing the control algorithm is less than half second. It is negligible compared to the control interval of 30 seconds.

5.2 APPLEware: Autonomic Performance and Power Control for Colocated Web Applications on Virtualized Servers

Today, popular Internet services have multi-tier and multi-service architecture in which various components invoke each other to process web requests [55]. Due to the complex inter-component performance dependencies and heterogeneity of application workloads, it is difficult to determine how many and what type of computing resources should be allocated to each service to achieve the performance assurance. The number as well as complexity of the applications being managed, have a significant impact on the agility and scalability of a performance and power management system. Furthermore, the contention of shared virtualized
infrastructure pose significant challenges in achieving the important goals of autonomic performance and power management.

We propose and develop, APPLEware, an autonomic middleware for joint performance and power control of co-located Web applications in virtualized computing environments. It dynamically allocates virtualized resources to various components of the hosted applications to meet their performance objectives in an agile and energy efficient manner. We use the terms power consumption and energy usage interchangeably since energy usage is measured over the same time period for all applications. Figure 5.6 shows APPLEware managing three multi-tier applications (App1, App3, App4) and one multi-service application (App2).

APPLEware’s core is a distributed model predictive control framework that scales well in large virtualized server systems. It applies machine learning based self-adaptive modeling of application behavior based on the performance and power measurements collected by sensors. The distributed controllers exchange information and co-operate with each other to tackle the important problem of performance interference between co-located applications. Virtualized resources are allocated to an application through the actuator.

APPLEware’s key design issues are:

1. **Autonomic performance control:** A self-managing middleware for a complex virtualized server system requires an automated method to monitor its operating environment; to analyze and model the complex system behavior; to plan a sequence of actions that achieve performance goals; and to execute those actions. It is challenging to achieve performance assurance in the face of highly dynamic and bursty workloads, complex interactions between application components, and performance interference be-
between co-located applications.

2. **Energy efficiency**: A common technique to reducing server energy consumption is to dynamically transition the hardware components from high power states to low-power states. However, changing the power state of a processor will affect the performance of multiple VMs running different applications in a virtualized computing environment. APPLEware achieves energy efficiency by controlling the virtualized resource usage of each VM, based on an accurate energy model.

3. **Scalability**: The computational complexity of an autonomic middleware increases significantly with the number as well as the complexity of the applications being managed. There are significant performance overheads in controlling a large system with many VMs spanning multiple server nodes. It is important but challenging to design a scalable middleware for autonomic performance and power management of large systems.

### 5.2.1 The Architecture

Figure 5.7 presents the APPLEware architecture. The computer system under control is a group of virtualized server nodes hosting multiple customer applications. We assume that each tier or service of a multi-tier multi-service application is deployed at a VM. Furthermore, VMs belonging to an application may span server nodes. An operator can specify the performance target and the priority of each application managed by APPLEware.

APPLEware employs a distributed control framework that decomposes the global performance and power
management problem of the entire system into local sub-problems for scalability. Each controller executes a control loop on a sub-system, which comprises of the VMs belonging to one application. We observe that some sub-systems are inherently coupled with each other mainly due to shared resource contentions and performance interference in a virtualized computing environment. APPLeware addresses this challenge through co-ordination among neighboring controllers for effective performance assurance and energy efficiency.

As an example, Figure 5.7 shows four distributed controllers. Each controller is responsible for the performance and energy efficiency of one application. Note that applications, app1 and app2, span three server nodes and share the underlying physical resources. The controllers, C1 and C2, regulate the resource allocation of VMs belonging to app1 and app2 respectively. At the same time, they co-ordinate with each other by exchanging information about their control decisions. Such co-ordination is important for the application performance assurance as well as system stability. This is due to the fact that a control action taken by C1 or C2 affects the performance of both app1 and app2. Similarly the controllers, C2 and C4, co-ordinate with each other to control app3 and app4.

Each autonomic controller performs the MAPE-K [51] control loop as follows:

1. **Monitor(M):** The performance and power monitors periodically measure the average end-to-end response time of the managed application and the average energy usage of the underlying multi-core server respectively. Our design does not use any semantic information regarding the performance metric. It treats the performance values as raw data for modeling and control. Hence, APPLeware is applicable to any performance metric.

2. **Analyze(A):** It constructs fuzzy models to analyze the complex system behavior in terms of the non-linear relationship between resource allocation and application’s end-to-end response time as well as energy usage. It also captures the coupling effects between neighboring applications that share the underlying physical resources.

3. **Plan(P):** It plans a sequence of control actions that regulate the allocation of virtualized resources for achieving the performance target and energy efficiency of the managed application. The control decisions are guided by Distributed Model Predictive Control theory.
4. **Execute(E)**: The optimization performed in the planning phase leads to a control action that brings the system closer to its performance target. The actuator executes the control actions in the form of adjustment in CPU and memory resources assigned to the application VMs.

### 5.2.2 System Modeling

#### 5.2.2.1 Global System Model

First, we consider a global system model that represents the performance and power consumption behavior of multi-tier and multi-service applications spanning across a group of virtualized server nodes. The inputs to the system are the resource allocation in terms of CPU and memory usage limits at various tiers and services of the hosted applications. The outputs of the system are the measured performance and average energy usage of each application. We obtain two separate models for power and performance of the system, respectively. The global system model is represented as follows:

\[
Y(k + 1) = F \cdot G(k) + H \cdot U(k) \tag{5.21}
\]

where \( U(k) \) is a vector of current resource allocations at sampling interval \( k \). \( H \) is a matrix that represents the impact of current resource allocations on the system outputs, \( Y(k + 1) \), at the next sampling interval. It also captures the coupling among applications due to performance interference in a shared virtualized environment. \( G(k) \) is a regression vector that contains the performance and power consumption values of each application in the current and previous sampling intervals. \( F \) is a regression matrix that represents the impact of regression vector on the system outputs.

Consider \( n \) applications in the system. The components of the global system model are described by the following equations:

\[
Y(k) = [y_1(k), y_2(k), \ldots, y_n(k)]^T \tag{5.22}
\]

\[
U(k) = [u_1(k), u_{c_1}(k), u_{c_1+c_2}(k), \ldots, u_{c_1+c_2+\ldots+c_n}(k)]^T \tag{5.23}
\]
In Eq. (5.22), each output term $y_i(k)$ represents the average end-to-end response time of application $i$ at sampling interval $k$. The output term for power modeling is the average energy usage of an application. In Eq. (5.23), each input term $u_j(k)$ represents the allocation of CPU and memory usage limits on a particular VM component $j$ in the virtualized server system. A VM component provides the functionality of a particular tier or service of a multi-tier or multi-service application. The total number of components in an application $i$ is denoted by $c_i$. The total number of VM components in the entire system is denoted by $C = \sum_{i=1}^{n} c_i$.

In the matrix $H$, the term $\eta_{i,j}$ represents the impact of resource allocation on the application performance or power consumption. $\eta_{i,j}$ has a non-zero value if the resource allocation input $u_j(k)$ has an impact on application $i$. In Eq. (5.24), $m_y$ specifies the order of regression, which is the number of previous samples of output variable $y_i$ that will be used in the system model. In the regression matrix $F$, the term $\zeta_{i,j}$ represents the impact of the system outputs measured at the $j_{th}$ previous sampling interval on the performance and power consumption of application $i$.

### 5.2.2.2 Problem Decomposition

Autonomic performance and power management of an entire virtualized server system containing many complex applications involves a large scale optimization and control process. The computational complexity of the problem grows significantly with the number of applications being managed. To address this scalability issue, APPLEnware decomposes the global performance and power control problem into a set of localized
subproblems.

From a local controller’s perspective, the goal of decomposition is to partition the set of system variables into three subsets, including local variables associated with the managed application, neighbor variables associated with other applications that have an impact on the performance and power consumption of the managed application, and all other variables in the system. The subproblem only includes its local and neighbor variables. The decomposition scheme reduces the number of variables involved in the control problem to improve system scalability. At the same time, it also captures the coupling among applications so that local controllers can achieve global system stability through coordination in their neighborhood.

The local variables in a control subproblem include a managed application’s performance, power consumption and the amount of virtualized resources allocated. The neighbor variables include the amount of resources allocated to the VM components of other applications that have a coupling effect on the local application due to shared resource contentions and performance interference. Note that a local controller does not control the neighbor variables. Instead, these variables influence the control decisions on the local subsystem.

The local model obtained by decomposing the global system model is described as:

\[ y_i(k + 1) = \zeta_i \xi_i(k) + \eta_i u_i(k) \]

Here, the output variable \( y_i(k) \) represents the performance or power consumption of application \( i \). \( \xi_i(k) \) and \( \zeta_i \) are subsets of regression vector \( G(k) \) and regression parameter matrix \( F \) respectively. They represent the current and previous outputs of application \( i \) and their impact on the application output in the next control interval. \( u_i(k) \) is a subset of vector \( U(k) \) that represents the allocation of CPU and memory resources on the VM components belonging to application \( i \) and the neighbor applications. \( \eta_i \) is a subset of matrix \( H \) that reflects the impact of resource allocation on the application output. As an example, a local controller \( C_1 \) in Figure 5.7 uses a local system model with the following parameters.

\[ u_1(k) = [u_1(k), u_2(k), u_3(k), u_4(k), u_5(k), u_6(k), u_7(k)]^T \]

\[ \eta(k) = [\eta_{1,1}, \eta_{1,2}, \eta_{1,3}, \eta_{1,4}, \eta_{1,5}, \eta_{1,6}, \eta_{1,7}] \]

From the controller \( C_1 \)’s perspective, \([u_1(k), u_2(k), u_3(k)]\) are a set of local variables, which represent three VMs of App1. \([u_4(k), u_5(k), u_6(k), u_7(k)]\) are the neighbor variables, which represent four VMs of App2 as shown in Figure 5.6. The local system model predicts the performance of App1 in the presence of
5.2.3 Fuzzy Modeling To Capture System Non-linearity

The prediction accuracy of the system model has a significant impact on the control performance. A linear model is often inadequate to accurately represent the complex behavior of inherently non-linear systems such as a multi-tier multi-service application hosted in a virtualized computing environment. APPLEware addresses this issue by constructing Fuzzy models that capture the performance and power consumption behavior of a local subsystem. The models include the local and neighbor variables of a subsystem according to APPLEware’s problem decomposition approach. A key strength of fuzzy model is its ability to represent highly complex and nonlinear systems by a combination of inter-linked subsystems with simple functional dependencies.

5.2.3.1 Model Formulation

A local sub-system is represented by a fuzzy model as follows:

\[ y_i(k+1) = R(\xi_i(k), u_i(k)). \] (5.28)

Similar to Eq. (5.25), \( y_i(k) \) is the output variable. \( u_i(k) \) consists of the local and neighbor input variables. The regression vector \( \xi_i(k) \) includes current and previous outputs of application \( i \).

\[ \xi_i(k) = [y_i(k), ..., y_i(k-m_y)]^T \] (5.29)

where \( m_y \) specifies the order of regression.

\( R \) is a rule based fuzzy model consisting of \( K \) fuzzy rules. Each fuzzy rule is described as follows:

\[ R_r: \text{If } \xi_{i1}(k) \text{ is } \Omega_{r,1} \text{ and } .. \text{ } \xi_{i\varrho}(k) \text{ is } \Omega_{r,\varrho} \text{ and } u_1(k) \text{ is } \Omega_{r,\varrho+1} \text{ and } .. \text{ } u_m(k) \text{ is } \Omega_{r,\varrho+m} \text{ then } \]

\[ y_i(k+1) = \zeta_r \xi_i(k) + \eta_r u_i(k) + \phi_r. \] (5.30)

Here, \( \Omega_r \) is the antecedent fuzzy set of the \( r_{th} \) rule which describes elements of regression vector \( \xi_i(k) \) and the current input vector \( u_i(k) \) using fuzzy values such as ‘large’, ‘small’, etc. \( \zeta_r \) and \( \eta_r \) are vectors containing the consequent parameters and \( \phi_r \) is the offset vector. \( \varrho \) denotes the number of elements in the regression vector \( \xi_i(k) \). The model output is calculated as the weighted average of the linear consequents in
the individual rules. That is,

$$y_i(k + 1) = \frac{\sum_{r=1}^{K} \beta_r (\zeta_r \xi_i(k) + \eta_r \upsilon_i(k) + \phi_r)}{\sum_{r=1}^{K} \beta_r}$$

(5.31)

where the degree of fulfillment for the \( r_{th} \) rule \( \beta_r \) is the product of the membership degrees of the antecedent variables in that rule. Membership degrees are determined by fuzzy membership functions associated with the antecedent variables. The model output is expressed in the form of

$$y_i(k + 1) = \zeta_r^* \xi_i(k) + \eta_r^* \upsilon_i(k) + \phi_r^*.$$  

(5.32)

The aggregated parameters \( \zeta_r^* \), \( \eta_r^* \) and \( \phi_r^* \) are the weighted sum of vectors \( \zeta_r \), \( \eta_r \) and \( \phi_r \) respectively.

### 5.2.3.2 Machine Learning Based Model Construction and Adaptation

APPLEware constructs initial fuzzy models by applying a subtractive clustering technique on performance and energy usage data collected from the system. Each obtained cluster represents a certain operating region of the system, where input-output data values are highly concentrated. The clustering process partitions the input-output space and determines the number of fuzzy rules and the shape of membership functions. APPLEware applies an adaptive network based fuzzy inference system (ANFIS) [53] to further tune the fuzzy model parameters.

Dynamic and bursty Internet workloads have significantly varying resource demands at multiple tiers and services of hosted applications. A static system model can not provide sufficient prediction accuracy of power and performance for all possible variations in the workload. APPLEware applies a wRLS (weighted Recursive Least Squares) method [72] to adapt the consequent parameters of its fuzzy models as new measurements are sampled from the system at runtime.

### 5.2.4 Distributed Control Design

Each local controller applies the distributed model predictive control principle to regulate a sub-system’s dynamic behavior towards the performance targets while minimizing the energy usage.
5.2.4.1 Control Formulation

A local control objective of controller $C_i$ is given by the following cost function:

$$ V_i(k) = \sum_{p=1}^{H_p} ||r_i - y_{i1}(k + p)||^2_P + \sum_{p=1}^{H_p} ||y_{i2}(k + p)||^2_Q + \sum_{c=0}^{H_c-1} ||\Delta u(k + c)||^2_R. \quad (5.33) $$

Here, $y_{i1}(k)$ is the average end-to-end response time and $y_{i2}(k)$ is the average power consumption of application $i$ at control interval $k$. The controller predicts both power and performance over $H_p$ control periods, called the prediction horizon. It computes a sequence of control actions $\Delta u_i(k), \Delta u_i(k + 1), ..., \Delta u_i(k + H_c - 1)$ over $H_c$ control periods, called the control horizon, to keep the predicted performance close to its pre-defined targets $r_i$ while minimizing the energy usage. The control action $u_i(k)$ is the change in CPU and memory usage limits imposed on various tiers and services of the multi-tier multi-service applications. $P$ and $Q$ are the tracking error weights that determine the trade-off between power and performance. The third term in Eq. (5.33) represents the control penalty and is weighted by $R$. This term penalizes big changes in control action and contributes towards high system stability.

The control problem is subject to the constraint that the sum of CPU and memory resources allocated to all VM components in the same physical server node must be bounded by the total CPU and memory capacity of the server.

For our experiments presented in Section 4.6, the value of $H_p$ was tuned to 20, which was sufficiently large for stable control. The value of $H_c$ was tuned to 5, which was able to provide good control performance. Due to space limitation, we did not include the sensitivity analysis.

5.2.4.2 Distributed Control Algorithm

APPLEware’s distributed control algorithm is shown in Algorithm 3.

5.2.4.3 Speeding Up Local Control

Although the decomposition of global system model into local subproblems reduces the computational complexity to a large extent, solving each local control problem still involves a non-convex and time-consuming optimization as formulated in Eq. (5.33). APPLEware addresses this issue by transforming each local control problem into a standard quadratic programming problem. For this transformation, it linearizes the fuzzy
Algorithm 3 Distributed Control Algorithm.
1: loop
2: A local controller $C_i$ measures the current state of the sub-system in terms of local and neighbor variables.
3: It compares the measured values of performance and power consumption of application $i$ with the predictions made by its Fuzzy models.
4: if A significant error in the prediction of performance and power consumption is detected then
5: It updates the Fuzzy models using its online learning algorithm.
6: end if
7: repeat
8: $C_i$ executes Model Predictive Control algorithm to solve local subproblem.
9: $C_i$ sends its control solutions to neighboring controllers. In addition, it receives the control solutions computed by its neighbors.
10: until The control solutions converge to a steady value.
11: It executes the control actions in the form of adjustment in CPU and memory resources assigned to the application $i$.
12: end loop
model at the current operating point and represent it as a state-space linear time variant model in the following form:

\[ x_i(k+1) = A(k)x_i(k) + B(k)u_i(k). \]

\[ y_i(k) = C(k)x_i(k). \]  

(5.34)

The state vector for the state-space description is defined as

\[ x_i(k+1) = [\xi_i^T(k), 1]^T. \]  

(5.35)

The matrices \( A(k), B(k) \) and \( C(k) \) are constructed by freezing the parameters of the fuzzy model at a certain operating point \( y_i(k) \) and \( u_i(k) \) as follows. First, we calculate the degree of fulfillment \( \beta_r \) for the current inputs (i.e., CPU and memory usage limits) chosen for the application and compute the aggregated parameters \( \zeta^*, \eta^* \) and \( \phi^* \). Comparing Eq. (5.32) and Eq. (5.34), the state matrices are computed as follows:

\[
A = \begin{bmatrix}
\zeta_{1,1}^* & \zeta_{1,2}^* & \cdots & \cdots & \zeta_{1,\rho}^* & \phi_1^* \\
1 & 0 & \cdot & \cdot & 0 & 0 \\
0 & 1 & \cdot & \cdot & 0 & 0 \\
\end{bmatrix},
B = \begin{bmatrix}
\eta_{1,1}^* & \eta_{1,2}^* & \cdots & \eta_{1,m}^* \\
0 & \cdot & \cdot & 0 \\
\cdot & \cdot & \cdot & \cdot \\
\end{bmatrix},
C = \begin{bmatrix}
1 & 0 & \cdot & \cdot & \cdot & 0 \\
\end{bmatrix}
\]

where \( \zeta_{ij}^* \) and \( \eta_{ij}^* \) are the \( j^{th} \) element of aggregate parameter vectors \( \zeta^* \) and \( \eta^* \) respectively for application \( i \).

The MIMO control problem defined by Eq. (5.33) is transformed to a quadratic program:

\[
\text{Minimize} \quad \frac{1}{2} \Delta u(k)^T H \Delta u(k) + c^T \Delta u(k) \]  

(5.36)

subject to constraint \( \Omega \Delta u(k) \leq \omega \).

The matrices \( \Omega \) and \( \omega \) are chosen to formulate the constraints on CPU and memory resource usage. Here, \( \Delta u(k) \) is a matrix containing the CPU and memory usage limits on each virtual machine over the entire control horizon \( H_c \). In the minimization formulation,

\[
H = 2(R_{1u}^T P R_{1u} + R_{2u}^T Q R_{2u} + R),
\]

\[
c = 2[R_{1u}^T P^T (R_{1v} A x(k) - r) + R_{2u}^T Q^T R_{2v} A x(k)]^T.
\]  

(5.37)
The matrices $R_{1u}, R_{1x}$ are associated with the performance models of hosted applications and matrices $R_{2u}, R_{2x}$ are associated with the power model of the resource pool.

$$
R_{iu} = \begin{bmatrix} C \\ CA \\ \vdots \\ CA^{H_p-1} \end{bmatrix}
$$

$$
R_{ix} = \begin{bmatrix} CB & 0 & \ldots & 0 \\ CAB & CB & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ CA^{H_p-1}B & CA^{H_p-1}B & \ldots & CA^{H_p-H_c}B \end{bmatrix}
$$

### 5.2.4.4 Computational Complexity Analysis

The computation overhead of APPLEware is dominated by the quadratic programming problem. Several optimization algorithms are available to solve a quadratic programming problem. We choose a widely used algorithm, the interior point method, to solve this problem. The algorithm has a computational complexity of $O(N)$ Newton iterations. Here $N$ is the number of decision variables that need to be computed to solve the given problem. Since each Newton iteration requires $O(N^3)$ algebraic operations, the worst-case computation complexity of the quadratic program solver is cubic in the number of decision variables. APPLEware is able to significantly reduce the computation overhead by decomposing the global control problem into local sub-problems. For APPLEware, the value of $N$ depends on the number of local and neighbor input variables only.

Furthermore, there are two important tunable parameters, $H_p$ and $H_c$ that affect the controller performance as well as the computation overhead. The prediction horizon $H_p$, in general, should be large enough such that the system converges to the reference target. However, the larger value of $H_p$ also increases the computation load. As the control horizon $H_c$ increases, the dimension of the solution space increases. Hence, the quality of control solutions will also improve. However, it has a significant impact on the time required to find the optimal solution.

In our control design experience, we observe that for the same value of $H_c$, a model predictive controller, in general, is able to achieve better control performance on a multi-tier application than on a multi-service
application. In other words, achieving effective control on the performance of a multi-service application requires a larger value of $H_c$. It is due to the fact that a multi-service application shows a very complex relationship between performance and the resource allocations at its various components. The situation is further worsened when there are multiple complex applications to be managed. This further strengthens our motivation to design and implement a distributed control framework that is scalable and agile for autonomic performance and power management.

### 5.2.5 Implementation

#### 5.2.5.1 Testbed

We built a testbed in a university prototype datacenter, which consists of Dell PowerEdge R610 servers. Each server has 2 Intel hexa-core Xeon X5650 CPUs and 32 GB memory. The servers are connected with 10 Gbps Ethernet. The testbed hosts multi-tier and multi-service Web applications as shown in Figure 5.6. Each tier and service of an application is implemented on a VMware virtual machine with 1 VCPU, 1 GB RAM and 15 GB hard disk space. All VMs use Ubuntu server 10.04 with Linux kernel 2.6.35. Each controller runs on a VM having 300 Mhz CPU usage limit and 128 Mb memory usage limit. We focus on using lightweight controllers that do not interfere with the performance of the hosted applications and that leave a small footprint on the energy usage of system.

For performance evaluation, we deploy the RUBiS benchmark in multi-tier and multi-service forms as shown in Figure 5.6. A multi-tier deployment of RUBiS has a simple pipelined architecture consisting of web, application and database servers. On the other hand, a multi-service deployment has a more complex architecture. In Figure 5.6, multi-service application App2 has a web service at the front end to handle the HTML requests. There are two application services, one for processing the requests that read from the database, another for processing all requests that write to the database. The database is implemented as a single shared service. In our implementation, a tier and service at the front end of the hosted application runs the Apache web server. The application tier and services run PHP servers. The database tier and the shared data service component runs a MySQL server.
5.2.5.2 APPLEware Components

We implemented the components of APPLEware as software modules that interact with each other as shown in Figure 5.7. APPLEware is deployed as a lightweight virtual appliance, which is distributed over separate machines on VMware infrastructure.

1. Power Monitor: The average energy usage of the virtualized server is measured at the VM level using VMware ESX 4.1. VMware gathers such data through its Intelligent Power Management Interface sensors. The power monitor module uses vSphere API to collect the energy usage data of a multi-tier multi-service application at each control interval. Energy usage is measured in terms of Kilo Joules (KJ).

2. Performance Monitor: APPLEware collects the application response time values from the Web-tier access logs, which are commonly available in typical e-commerce applications. We inject an XML-RPC daemon program that runs at the web tier to measure the average end-to-end response time of requests. APPLEware’s performance monitor module consists of an XML-RPC client that communicates with RUBiS application to periodically collect the performance statistics at each control interval.

3. Performance and Power modeling: We use MATLAB’s Fuzzy Logic Toolbox to apply subtractive clustering and ANFIS modeling technique on the data collected from the server system. At runtime, the performance and power models are updated according to new measurements collected from the system using the wRLS algorithm.

4. Distributed Controller: Each controller module invokes a quadratic programming solver, quadprog, in MATLAB to compute the local control solution. We used MATLAB Builder JA to create a Java class
from the MATLAB program invoking *quadprog*. This Java class is integrated into APPLEware source code and deployed to each local controller node. The distributed controllers communicate with each other in a peer-to-peer manner using XML-RPC protocol.

5. Actuator: It uses vSphere API to impose CPU and memory usage limits on the VMs. The vSphere module provides an interface to execute a method `ReconfigVM_Task` to modify a VM’s resource usage limit.

The performance overhead of APPLEware’s distributed controllers is mainly affected by three factors: (1) time taken to collect performance statistics from the applications, (2) time required to compute a control decision, (3) actuation time. Figure 5.8 shows the average time taken for each of these factors on our testbed hosting four applications. Note that control overheads play an important role in determining the control interval. We set the control interval of APPLEware’s distributed controllers to be 10 seconds, which is sufficiently large to overcome the control overheads and also avoid measurement noise.

5.3 Evaluation

5.3.1 PERFUME

5.3.1.1 Power and Performance Assurance with Flexible Tradeoffs

A key feature of PERFUME is its ability to assure joint power and performance guarantee with flexible tradeoffs while assuring control accuracy and system stability. The tradeoffs between inherently conflicting power and performance objectives can be specified by a datacenter administrator. The system stability is measured in terms of relative deviation of power and performance from their respective targets, as defined in vPnP [43]. The relative deviation is $|y(k) - r|/r$, where $y(k)$ is the measured application performance or the power consumption of the resource pool at time interval $k$ and $r$ is the corresponding target value. We experiment with power-preferred, performance-preferred and balanced control options under a highly dynamic workload [70]. Figure 5.9(a) shows the dynamic changes in the number of concurrent users. The workload prediction component is not used in this case, due to the difficulty in achieving sufficient prediction
PERFUME achieves the specified tradeoffs by tuning the tracking error weights, $P$ and $Q$, in the MIMO control objective defined by Eq. (5.33). Figure 5.9(b) compares the control accuracy of vPnP with PERFUME in assuring the throughput target for various tradeoffs between power and performance. Our results demonstrate that, compared to vPnP, PERFUME delivers average improvement of 30% in performance assurance in terms of relative deviation for various control options. We obtained similar results with the average improvement of 25% for relative deviation in power consumption with respect to its power budget, as shown in Figure 5.9(c). The control accuracy of the power-preferred option is the highest for power assurance but the lowest for throughput assurance. Whereas, the control accuracy of the performance-preferred option is the highest for throughput assurance and the lowest for power assurance. The balanced control option shows good control accuracy for both power and performance assurance.
5.3.1.2 System Stability

We now take a closer look at the system stability of PERFUME under the highly dynamic workload. We experiment with the power-performance balanced control option. Figures 5.10(a) and 5.10(b) illustrate that PERFUME offers more accurate assurance of power and performance targets compared to vPnP [43]. We show the results for only one of the hosted RUBiS applications. Similar results were obtained for the other. PERFUME is able to adapt itself and control both power consumption and throughput so that they eventually converge to the steady state. On the other hand, results show there are more significant oscillations in power and performance assurance due to the lack of control accuracy and system stability guarantee in vPnP. There is an improvement of 25\% and 32\% in relative deviation of power consumption and throughput respectively.

Figure 5.10(c) compares the total CPU usage limits allocated by vPnP and PERFUME at various sampling intervals. The total CPU usage limits is the sum of the CPU usage limits at the Web, application and database tiers. On average, PERFUME uses similar amount of CPU resources as vPnP. However, there is significantly less fluctuations in resource allocation. The CPU allocations are adjusted to track the power and performance targets. For instance, when an increase in the workload intensity causes the power consumption to exceed the power budget, CPU allocation is reduced to bring the power consumption close to the power budget. The modeling accuracy, self-adaptiveness and control theoretic foundation of FUMI control enables PERFUME to achieve system stability and accurate control for both power and performance of multi-tier applications in the face of highly dynamic workloads.
5.3.1.3 Percentile-Based Response Time Guarantee

We now demonstrate the capability of PERFUME in accurately achieving the 95\textsuperscript{th}-percentile response time guarantee in the face of highly dynamic workload. Note that PERFUME is able to provide any percentile based response time guarantee. Compared with the mean-based performance metric, a percentile-based response time introduces much stronger nonlinearity in the system. Figure 5.11 shows that compared to vPnP [43], PERFUME delivers significantly improved control accuracy and performance assurance for highly non-linear percentile-based response times. For this experiment, we set the 95\textsuperscript{th}-percentile response time target of a RUBiS application as two seconds. We observe that compared with vPnP, there is the improvement of 40\% in terms of relative deviation by PERFUME. This is mainly due to two reasons. First, PERFUME’s FUMI model has better accuracy, even in case of highly non-linear percentile-based performance. Second, it provides more accurate control and system stability due to the FUMI control design.

5.3.1.4 System Robustness under A Bursty Workload

We evaluate the robustness of PERFUME under a bursty workload. We use an approach proposed in [98] to inject burstiness into the arrival process of RUBiS clients according to the index of dispersion. The dispersion index modulates the think times of users between submission of consecutive requests. We set the index of dispersion to 4000 and the maximum number of concurrent users to 1000. Figure 5.12 (a) shows the bursty workload in which the number of active users in a RUBiS application fluctuates over a period of 200 seconds. Figures 5.12(b) and 5.12(c) illustrate that, compared to vPnP, PERFUME is able to provide better assurance of average power consumption and throughput targets in the face of the bursty workload. We choose a sampling interval of 20 seconds for both approaches. A smaller sampling interval provides better responsiveness.
to workload fluctuations, but increases the sensitivity towards random noise. The variations in the average power consumption and throughput are mainly due to burstiness in the workload and the control actions (CPU allocations) taken at each sampling interval. The robustness of PERFUME under bursty workloads is attributed to the fact that its control actions are based on a more accurate model of the system and a sound control theoretic foundation. Moreover, PERFUME is more adaptive to variations in workload due to its fast online learning algorithm. We observe that compared with vPnP, there is the improvement of 32% and 44% in terms of relative deviation of power and throughput by PERFUME.

5.3.1.5 Impact of Proactive FUMI Control on Power and Performance Assurance

We demonstrate the benefit of integrating workload prediction with FUMI control. We modify the RUBiS client to generate workload based on the Web traces from the 1998 Soccer World Cup site [11]. These traces contain the number of arrivals per minute to this website over an 8-day period. As a case study, we choose the workload trace of a moderately busy day and compress the original 24-hour long trace to 1 hour, similar to the related work in [123]. Figure 5.13 shows the actual workload and the predictions made by Kalman filtering technique. Our proactive FUMI controller acquires workload predictions over a horizon of four
steps and computes the CPU resource adjustments required to meet the power and performance goals. The power budget and the performance target are set to be 15 Watts and 2500 requests per sampling interval respectively.

Figures 5.14(a) and 5.14(b) compare the average power consumption and system throughput achieved by PERFUME with and without the integration of workload prediction. In case of a purely reactive FUMI control, we observe significant violations of the power and performance targets during the time intervals 10-13 min, 21-29 min and 34-41 min. It is due to the continuous increase in the workload intensity during those intervals as shown in Figure 5.13. Without the integration of workload prediction, FUMI control incorrectly estimates the future states of the system assuming that the workload intensity will not change. Furthermore, it requires a few control intervals to update the system model in response to the workload variations. As a result, the control actions in terms of CPU resource adjustments lag behind the continuously increasing workload.

On the other hand, the integration of workload prediction allows FUMI control to make proactive control decisions in anticipation of future changes in the workload intensity. Hence, it is able to avoid significant violations of power and performance in the face of dynamic workload variations. For instance, Figure 5.14(a) shows that the average power consumption starts decreasing at time 18 min due to proactive control action in anticipation of future workload change starting at time 21 min. As a result of this trend, the controller is able to keep the power consumption close to its target during the time interval 21-29 min in spite of the continuous increase in the workload. Figure 5.14(c) shows the improvement achieved by proactive FUMI control in terms of the relative deviation of power and performance.

5.3.1.6 Service Differentiation Provisioning

We evaluate the service differentiation capability of PERFUME in the face of a dynamic power budget as shown in Figure 5.15(a). In practice, the power budget might change due to thermal condition or temporary reductions in cooling or power delivery capacity.

In this experiment, two RUBiS applications, app1 and app2, are hosted on a shared resource pool. For service differentiation, the performance tracking error weight $Q$ for app 1 is set to be larger than that of app
2. Hence, PERFUME gives a higher priority to app 1. As a case study, a power-preferred control policy is applied. Each application faces a workload of 1000 users. The throughput target is set to be 4000 requests per sampling interval.

Figures 5.15(b) and 5.15(c) show the average power consumption of the resource pool and the achieved throughput of the RUBiS applications. For the first 39 intervals, both applications are able to meet their performance goals while keeping the average power consumption below the specified power budget of 25 Watts. At interval 40, the power budget is changed to 15 Watts. This constrains the system in such a way that both applications can not maintain their current performance level without violating the power budget. PERFUME responds to this situation automatically and correctly re-distributes CPU resources so that the higher priority application, app1, still achieves an average throughput that is close to the target. On the other hand, the performance of lower priority application is degraded. As a result of this service differentiation, PERFUME is able to avoid power budget violations.

5.3.1.7 Effect of Control Parameter Tuning

We now present the effect of tuning FUMI control parameters on the control performance. Figure 5.16 shows the performance of one RUBiS application under a workload of 1000 concurrent users, for different values of control penalty weight $R$. For $R = 0.02$, the controller adjusts the CPU resources more quickly and aggressively to meet the performance target. However, it results in large oscillations in performance. On the other hand, for $R = 0.06$ the controller avoids significant oscillations in performance. However, it becomes much more sluggish and takes ten control intervals to meet the performance target. We observe that a control penalty weight $R = 0.04$ balances the tradeoff between control stability and responsiveness.
Next, we study the impact of tuning the prediction horizon $H_p$ on the control performance. In this experiment, the RUBiS application faces a dynamic workload shown in Figure 5.13. The control performance is measured in terms of the relative deviation of average throughput with respect to the performance target of 2500 requests per control interval. Figure 5.17 shows that the control performance initially improves with the increase in $H_p$. Intuitively, as the controller looks further ahead, it can anticipate future workload demands and start taking control actions accordingly at the current time step itself. However, control performance degrades when $H_p$ is larger than four. It is due to the fact that the prediction accuracy decreases with the increase in the prediction horizon. Therefore, $H_p$ must be chosen carefully, considering the trade-off between look-ahead performance and estimation errors.
Table 1: APPLEware’s model validation for the multi-service application (App2).

<table>
<thead>
<tr>
<th>Service1</th>
<th>Service2</th>
<th>Service3</th>
<th>Service4</th>
<th>Measured resp. time</th>
<th>Predicted resp. time</th>
<th>Measured energy</th>
<th>Predicted energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>400 Mhz</td>
<td>1100 Mhz</td>
<td>500 Mhz</td>
<td>800 Mhz</td>
<td>2532 ms</td>
<td>2658.6 ms</td>
<td>35 KJ</td>
</tr>
<tr>
<td>Mem</td>
<td>768 Mb</td>
<td>256 Mb</td>
<td>512 Mb</td>
<td>1024 Mb</td>
<td>35 KJ</td>
<td>37.2 KJ</td>
<td></td>
</tr>
<tr>
<td>CPU</td>
<td>400 Mhz</td>
<td>700 Mhz</td>
<td>400 Mhz</td>
<td>1300 Mhz</td>
<td>1669 ms</td>
<td>1777.4 ms</td>
<td>31.2 KJ</td>
</tr>
<tr>
<td>Mem</td>
<td>256 Mb</td>
<td>128 Mb</td>
<td>768 Mb</td>
<td>768 Mb</td>
<td>29 KJ</td>
<td>30 KJ</td>
<td></td>
</tr>
<tr>
<td>CPU</td>
<td>500 Mhz</td>
<td>768 Mb</td>
<td>512 Mb</td>
<td>512 Mb</td>
<td>1304 ms</td>
<td>1277.9 ms</td>
<td>30 KJ</td>
</tr>
<tr>
<td>Mem</td>
<td>512 Mb</td>
<td>1024 Mb</td>
<td>512 Mb</td>
<td>1024 Mb</td>
<td>29 KJ</td>
<td>29 KJ</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.18: APPLEware's performance prediction (APP1) in the presence of interference effects.

5.3.2 APPLEware

5.3.2.1 Model validation

The accuracy of the system model has a significant impact on effective control of power and performance. We first validate APPLEware’s system models using a multi-tier application App1 and a multi-service application App2. Note that various service components of App2 are co-located with VMs belonging to App1. The initial models are obtained by using a training data set that consists of the average end-to-end response time and energy usage measurements of the two applications subject to randomly varying CPU and memory usage limits. Each application faces a workload of a browsing mix of 600 concurrent users. For model validation, we use a different set of resource allocations that is not used for training the system models.

Table 1 shows the performance and energy usage prediction results of APPLEware for App2. In this experiment, we statically allocate 1000 GHz CPU and 512 MB memory at each VM component of the co-hosted App1. We compare the measured and predicted values of average end-to-end response time and average energy consumption over a period of one hour for various CPU and memory allocations on App2’s
VMs. Both prediction errors for performance and energy usage are under 7% in all cases, which confirms the accuracy of APPLEware’s model.

Next, we demonstrate APPLEware’s performance prediction accuracy in the presence of interference between co-located applications. Figure 5.18 shows the variations in the average end-to-end response time of App1 due to the interference caused by various resource allocations on App2. Note the resources allocated to App1 remains fixed. APPLEware is able to accurately predict application performance with a small normalized root mean square error (NRMSE) of 9%. NRMSE is a standard metric for deviation.

APPLEware’s prediction accuracy is due to its fuzzy modeling that captures the inherently non-linear relationship of performance and energy with multiple virtualized resources while considering the impact of performance interference among co-located VMs. We observe that our fuzzy models use 14 fuzzy rules to represent the performance of the multi-service application, App2, with sufficient prediction accuracy. Whereas, the models use only 9 fuzzy rules in case of multi-tier application, App1. It is due to the fact that multi-service applications exhibit more complex relationship between performance and resource allocations. Hence, they require more complex models for performance prediction.

5.3.2.2 Autonomic Performance Control and Energy Efficiency

Control Agility We evaluate the effectiveness and agility of APPLEware in assuring the performance and reducing the energy usage of co-located multi-tier and multi-service applications. As a performance metric, we use the average end-to-end response time, which represents the user perceived performance of interactive Internet applications. The SLA target for all the applications is set to 1200 ms. We apply a stationary
workload of 600 concurrent users to each application. Figure 5.19(b) shows that APPLEware is able to bring the average end-to-end response time of each application close to their respective SLA targets within 60 seconds. It is due to the agile and effective adjustments in the CPU and memory resources of the multi-tier multi-service applications by APPLEware’s distributed controllers. On the other hand, Figure 5.19(a) shows that a centralized controller takes around 100 seconds to meet the performance target.

For any system, a centralized controller at the fastest sampling rate gives the best achievable performance. However, implementing centralized controller at the fastest sampling rate may not be feasible, due to operational constraints such as the overheads involved in measuring the current system states and computing the control decisions. In such cases, a distributed control approach presents an opportunity to obtain superior control performance. In this experiment, we found the worst-case control overhead of the centralized controller to be seven seconds. As a result, its control interval needs to be much larger than APPLEware’s control interval of 10 seconds. Hence, APPLEware provides better control agility than a centralized controller. Furthermore, the control solutions of APPLEware’s distributed controllers are able to converge close to the optimal solutions obtained by the centralized controller. Note that the convergence takes place within each control interval.

To quantify the performance of APPLEware, we use the relative deviation from a target as the metric. The relative deviation for performance is \(|y(k) - r|/r\), where \(y(k)\) is the average end-to-end response time of an application at time interval \(k\) and \(r\) is the SLA target for that application. Figures 5.20(a) and 5.20(b) show that APPLEware is able to improve the relative performance deviation as well as the energy efficiency of each application, compared to the centralized controller. On average, the improvement in the relative deviation by
APPLEware is 37%. It is also 12% more energy efficient than the centralized controller. The improvement in energy efficiency is due to the fact that APPLEware drives the system towards optimal operating conditions more quickly than the centralized controller does.

Robustness under dynamic and bursty workloads  Next, we evaluate the robustness of APPLEware under dynamic and bursty workloads. As a case study, we apply a step-change workload as shown in Figure 5.21(a) to App1 and App3. On the other hand, we apply a bursty workload to App2 and App4. We inject burstiness into the arrival process of RUBiS clients according to the index of dispersion. The dispersion index modulates the think times of users between submission of consecutive requests. We set the index of dispersion to 4000 and the maximum number of concurrent users to 1000. Figure 5.21(b) shows a bursty workload.

Figures 5.22(a) and 5.22(b) compare the performance assurance capability of APPLEware with that of PERFUME in the face of the dynamic and bursty workloads. In Figure 5.22(a), both PERFUME and APPLEware show some fluctuations in the average response times at the control intervals 17 to 21 and 34 to 38. It is due to the abrupt changes in the workload starting at control interval 17 and 34. However, APPLEware is able
to be able to adapt itself more effectively so that the average end-to-end response time of App 1 converges to the
SLA target of 1000 ms within few control intervals. The fluctuations in the average response time of App2 is
more significant mainly due to burstiness in the workload, which is more difficult to handle. However, com-
pared to PERFUME, APPLEware is able to keep the response time closer to the SLA target. The robustness
of APPLEware under dynamic and bursty workloads is attributed to its fast online learning algorithm, which
updates the performance model by observing dynamic system behavior. Importantly, self-configuration of
APPLEware is effective due to its awareness of the performance interference effects.

Figures 5.23(a) and 5.23(b) show the average relative deviations and the energy usage of various multi-
tier and multi-service applications under the dynamic and bursty workloads. Compared with PERFUME,
there is the improvement of 49% on average in terms of relative deviation by APPLEware. At the same time,
APPLEware improves the energy efficiency by 20%. We observe that both relative deviation and energy
usage of App2 and App4 are larger than that of App1 and App3. It is due to the fact that these applications
face a bursty workload, which demonstrate abrupt workload variations. However, APPLEware still shows
significant improvement in performance and energy efficiency compared with PERFUME.

5.3.2.3 Scalability Analysis

For scalability analysis, we build a testbed of a virtualized computing environment as shown in Figure 5.24.
The system has seven physical servers running a total of 29 VMs hosting nine multi-tier multi-service appli-
cations. A VM running the $j_{ih}$ tier or service of an application $i$ is denoted by $A_{ih}$. APPLEware uses nine
distributed and lightweight controllers. Each controller $C_i$ runs on a VM having only 300 Mhz CPU usage
We measure the performance overhead and energy usage of the controllers themselves when the total number of applications in the system are increased. Figure 5.25(a) shows that the per-controller execution time of APPLEware becomes significantly smaller than that of the centralized controller with increasing number of applications. We observe that in cases of two and three applications, the overhead of APPLEware is slightly larger than that of the centralized controller. It is because the overhead caused by the co-ordination among APPLEware’s distributed controllers overshadows the pure computation overhead, when a small number of applications are being managed.

Figure 5.25(b) compares the total energy usage of APPLEware’s controllers themselves with that of the centralized controller. The energy usage of the centralized controller increases rapidly when the number of applications being managed are increased. On the other hand, APPLEware’s energy usage shows a gradual increase. It is due to the fact that the total energy usage is dominated by the execution time of the controllers. Energy usage is a product of the average power consumption and the execution time of the controllers.

The evaluation and analysis demonstrate APPLEware’s superior scalability, which is mainly due to its distributed control framework.

5.4 Summary

Datacenters face significant multi-facet challenges in power and performance management for meeting SLAs, resource utilization efficiency and power savings. PERFUME provides a coordinated and self-adaptive power and performance control in a virtualized server cluster. As demonstrated by experimental results based on a
Figure 5.25: APPLEware’ controller overhead with increasing number of applications.

(a) Per-controller execution time. (b) Total energy usage of the controllers.

testbed implementation, its main contributions are the precise and proactive control of power consumption of virtualized servers for power budgeting, average throughput and percentile-based response time guarantee of multi-tier applications, tradeoff flexibility between power and performance targets and service differentiation among co-located applications while assuring control accuracy and system stability in the face of highly dynamic and bursty workloads. The main technical novelty of PERFUME system is due to the proposed fuzzy MIMO (FUMI) control technique, which integrate the strengths of fuzzy logic, MIMO control and artificial neural network. It is self-adaptive to dynamic workloads due to online learning of fuzzy model parameters using a computationally efficient wRLS method. This is complemented by the integration of future workload prediction for proactive control.

The increasing scale and complexity of virtualized server systems hosting multi-tier multi-service applications pose significant challenges to autonomic performance and power management. APPLEware is an autonomic and scalable middleware for joint performance and power control in virtualized computing environments. It is easily deployable as a lightweight virtual appliance on VMware infrastructure. As demonstrated by modeling, analysis and experimental results based on testbed implementation, its main contributions are robust performance assurance, energy efficiency and scalability in the presence of VM performance interference as well as highly dynamic and bursty workloads.
Chapter 6

Power-Aware Placement and Migration in High Performance Computing Datacenters

Today, some of the fastest supercomputers in the world rely on GPUs to advance scientific discoveries. However, the proliferation of power-hungry GPU accelerators in High Performance Computing datacenters impose new challenges in power management.

Traditionally, GPU-accelerated application executions are tightly coupled to the physical GPU hardware, requiring each computational node to be equipped with one or more local GPUs. Recent efforts on GPU virtualization such as VOCL [142] and rCUDA [33] expose physical GPUs as decoupled virtual resources. Virtualization enables better management of GPU resources for improved resource utilization and fault tolerance. However, the potential use of GPU virtualization for power management in GPU-enabled server clusters is not well-explored. We investigate and enable dynamic scheduling of GPU resources for online power management in virtualized GPU environments.

We present a power-aware virtual OpenCL (pVOCL) framework that controls the peak power consumption and improves the energy efficiency of GPU-enabled server clusters. It automates power management in datacenter cabinets by performing dynamic consolidation and power-phase topology aware placement of GPU workloads. The pVOCL runtime system periodically checks the configuration of server clusters in the face of dynamically varying GPU resource demands. Then, it performs an optimal sequence of adaptation actions that will drive the system towards the most power-efficient configuration, without violating the given power budget. The adaptations involve changing the power states of compute nodes and performing live
migration virtual GPUs. For GPU virtualization and virtual GPU migration capability, it utilizes the VOCL framework. However, our power-aware dynamic placement and migration approach is applicable to other GPU virtualization frameworks as well.

### 6.1 GPU Virtualization

First, we present an overview of an existing GPU virtualization framework, VOCL [142]. VOCL is based on the OpenCL programming model and uses Message Passing Interface (MPI) for data communication across different computing nodes when remote GPUs are used. VOCL exposes the same API as the OpenCL programming model.

Figure 6.1 shows VOCL components.

Figure 6.1: The Virtual OpenCL (VOCL) framework.

The VOCL library is located on the local node. It is responsible for forwarding the OpenCL function calls to the corresponding GPUs and returning the GPU outputs to the application. It calls the native OpenCL functions to perform GPU computation on local GPUs. For using a remote GPU, it wraps up the inputs of the OpenCL function and sends them to the remote computing node using MPI communication.

The VOCL proxy is located on the remote node. It is a service provider for applications to use GPUs in the node. It receives MPI connection requests from the VOCL libraries and establishes communication channels with them. It is responsible for decoding the messages received from VOCL libraries, calling the native OpenCL functions to perform computation and sending back the function output to the VOCL libraries.

VOCL also supports live task migration across different computing nodes. Such migration is achieved
by changing the mapping relationship between *virtual GPUs* (or *VGPU*) in the VOCL library and the VOCL proxy. Specifically, a VGPU represents the resources used by an application on each physical GPU. It includes the OpenCL resources such as the context, device memory, and kernels. VGPUs exist in both the VOCL library and the VOCL proxy, which are referred to as *VOCL VGPU* and *OpenCL GPU*, respectively. The VOCL framework has a one-to-one mapping between the VOCL VGPU and the OpenCL VGPU.

### 6.2 GPU Consolidation and Power-phase Awareness

The most common power generation and distribution system in use today is the balanced, three-phase system. This system is comprised of three equal-amplitude, sinusoidal voltages, that are offset from one another by $120^\circ$ phase. As a consequence of this configuration, the instantaneous voltage across the phases sums to zero, which translates into mechanical balance and greater efficiency for rotating power generation machinery.

A critical factor in the efficiency of the three-phase system is the balance of the load across phases. An imbalance in current across phases results in lost power through neutral line current. We measure the efficiency with respect to three-phase load configuration in Figure 6.2, using the experimental platform described in Section 6.4. We compare the power consumption of several node configurations with a configuration where the load is balanced equally across all three phases. Six compute nodes are used and we measure the power consumed over several permutations of these nodes with respect to power phase. The phase imbalance of a three-phase system is expressed as a percentage value, often defined as the maximum deviation from the average of the three-phase voltages or currents, divided by the average of the three-phase voltages or currents. We use the power consumption measurement to quantify the phase imbalance in place of voltage or current.
Figure 6.3: The original system configuration and two possible resulting configurations after VGPU migration and consolidation.
Given the impact of balancing the workload across phases, we explore two possible configurations for a job executing on our cluster system, shown in Figure 6.3. In the original configuration, two compute nodes (i.e., Proxies 2 and 3) both have two physical GPUs, but only one is currently mapped to a virtual GPU. We consolidate these two virtual GPUs to a single node, leaving one node idle and allowing it to be powered down to save power. In Figure 6.3(a), one way of GPU consolidation results in an unbalanced configuration with a phase imbalance of 4. In Figure 6.3(b), another way of GPU consolidation results in a balanced configuration with a phase imbalance of 0.

Figure 6.4 shows the total energy and peak power consumption for the original configuration and each configuration after consolidation. This data was measured using power monitoring equipment on our experimental cluster. We observe that consolidation yields a significant improvement in total energy, as well as a corresponding reduction in the peak power consumption. In addition, the balanced configuration achieves higher efficiency, with a 7.6% reduction in energy and 5.5% reduction in the peak power.

Figure 6.5 demonstrates the impact of GPU workload placement on the power consumption. Assuming a resource demand for two GPUs, the power consumption varies significantly depending on how the GPU workloads are distributed across compute nodes and CDU power phases.
6.3 pVOCL: Power-Aware Dynamic Placement and Migration in Virtualized GPU Environments

6.3.1 pVOCL Framework

Figure 7.2.1 presents an architectural overview of the management components used in our power-aware VOCL framework. It consists of a power model, a topology monitor, a power optimizer, and a migration manager. The power model captures the power consumption trends for various configurations of VOCL proxy nodes in datacenter cabinets. The topology monitor periodically collects the information regarding the configuration, conf(k), at the current control interval k. The power optimizer determines the optimal configuration that minimizes the total power consumption of the system while considering the overheads involved in the adaptation from the current configuration to the new configuration. The migration manager performs node reconfiguration of VOCL proxy nodes and live migration of existing VGPUs. Together these
6.3.2 Power Modeling

6.3.2.1 Power-Phase Awareness

In order to make the most power-efficient GPU consolidation and node placement decision, a model is needed that captures the impact of various node configurations on the overall power consumption. A node configuration encapsulates the mapping of VGPUs to VOCL proxy nodes and the placement of VOCL proxy nodes in the datacenter cabinets in order to minimize the power consumption.

First, we measured the total power consumption by turning on idle nodes at various power phases of the two cabinets as shown in Figure 6.7. The notations $n_1n_2n_3$ and $n_1n_2n_3-n_21n_22n_23$ represent node
configurations in one cabinet and two cabinets respectively. $n_{ij}$ denotes the number of nodes turned on in the $i_{th}$ cabinet and the $j_{th}$ power-phase. The power consumption increases as more nodes are turned on, as expected. For a given number of nodes, we observe a significant variation in the power consumption for different node placements. A node configuration that is more balanced in terms of the number of nodes turned on at each power-phase is more power-efficient than other configurations.

Next, we examined whether the power consumption trend can be generalized to active nodes that execute GPU workloads. We analyzed the power profiles of four application kernel benchmarks: (matrix-multiplication (MM), n-body (NB), matrix-transpose (MT), and Smith-Waterman (SWAT). The first two kernels are compute-intensive; the other two require more data movement between host memory and device memory. In this experiment, each node configuration used two GPUs in total. As indicated in Figures 6.8(a) and 6.8(b), all four application kernels show similar variations in the power consumption and energy usage for different node configurations. Figure 6.8(c) shows that the application performance is independent of the node configuration.

We note that a generalized model that captures the power consumption trends for various node configurations provides sufficient information to find the most power-efficient GPU consolidation and node placement. The power minimization problem does not need actual power numbers corresponding to various application kernels. Hence, we design pVOCL to use a simple lookup table-based power model. The lookup table is generated based on the power usage data for various node configurations as provided by the datacenter administrator.

6.3.2.2 Analysis of Reconfiguration Overhead

A node reconfiguration of a virtualized GPU environment involves three types of actions. They are to turn on a new compute node, migrate existing VGPUs across different nodes and to turn off a compute node. The time required by a newly powered on compute node to be ready for computation may take a few seconds to a few minutes depending on the computing platform. However, it does not impact the application performance in our pVOCL framework. It is due to the fact that the GPU workloads continue to execute in the existing nodes at the time of node reconfiguration. The workloads are only migrated when the new compute nodes
We analyze the performance overhead caused by VGPU migration. In Figure 6.9 shows the total execution time for various application kernel benchmarks with no migration and when a single migration is performed. We observe that, overall, as the problem size increases, the relative performance overhead due to migration decreases. It is due to the fact that the execution time increases faster than the migration overhead with regard to the problem size. In addition, the migration overhead is a few hundred milliseconds for compute-intensive kernels and up to a few seconds for memory-intensive kernels. We conclude that the performance degradation caused by migration can be negligible for programs running a reasonably long time (e.g., a few tens of seconds). Furthermore, note that we use Ethernet connected compute nodes in our experiments. The use of high speed InfiniBand can further reduce the migration overhead drastically.

### 6.3.3 pVOCL Components Design

GPU-enabled server clusters in a datacenter face dynamically varying GPU resource demands from multiple users. As a result, the number of GPUs used in various nodes changes over time. Furthermore, the mapping of nodes to different power phases in a cabinet may also change as new nodes are installed or old nodes are removed for system upgrade and maintenance. The topology monitor periodically communicates with the CDUs and the VOCL proxy nodes to collect the information about the current node configuration. Note that the VOCL proxy does not distinguish between the communication messages from the topology monitor and an application host. It merely receives the MPI messages and replies back after performing certain actions according to the information provided in the MPI message tag. We implemented the interfaces necessary in the VOCL proxy module to enable topology monitoring.

#### 6.3.3.1 Topology Monitor

As Figure 6.10 shows, the topology monitor performs the following operations.

1. Utilize the remote monitoring capability provided by the CDUs to get the number of nodes that are turned on at various power phases of the datacenter cabinets. The communication is performed by using a telnet interface to each CDU.
2. Call the `MPI_Comm_connect()` function to establish communication channel with each VOCL proxy node. The information regarding existing VOCL proxy nodes are stored and updated locally by the topology monitor.

3. Send control messages to the VOCL proxy nodes to request for all the GPU device information, including the number of available GPUs and the device Ids. For this purpose, it calls the `MPI_Isend()` function with the MPI message tag `GET_PROXY_INFO`.

4. Call `MPI_Irecv()` to receive GPU device information from the VOCL proxy nodes.

5. Call `MPI_Isend()` with the MPI message tag `MAP_INFO` to request the information about various VGPU to physical GPU mappings. Note that a physical GPU that is not mapped to any VGPU indicates that it is not used by any application. We assume that at most one VGPU is mapped to a physical GPU.

6. Call `MPI_Irecv()` to receive VGPU mapping information from the VOCL proxy nodes.

### 6.3.3.2 Power Optimizer

GPU consolidation and placement in a virtualized GPU environment may involve a sequence of adaptation steps that include turning on new compute nodes, migrating VGPUs and turning off compute nodes across different power-phases of datacenter cabinets. A direct transition between two node configurations may not be possible due to potential power budget violations. We formulate dynamic GPU consolidation and placement as the following optimization problems.
Optimization 1:

Minimize \( p(c_n) \) \hspace{1cm} (6.1)

Subject to Constraints:

for all \( c_i \epsilon [c_1, c_2, ..., c_n] \) \hspace{1cm} (6.2)

\[ p(c_i) < P \] \hspace{1cm} (6.3)

\[ g(c_i) = g(c_0) \] \hspace{1cm} (6.4)

Here, \( p(c_i) \) is the total power consumption of all the nodes used in configuration \( c_i \). \( c_1, ..., c_{n-1} \) are the intermediate configurations generated by applying GPU consolidation and node placement actions, starting from the initial configuration \( c_0 \). Let \( g(c_i) \) denote the total number of GPUs used for the configuration \( c_i \). The objective Eq. (6.1) is to find a set of target node configuration, \( c_n \), that minimizes the total power consumption while satisfying some constraints. Eq. (6.2) and Eq. (6.3) state that the power consumption due to an intermediate configuration must not violate the given power budget \( P \). Eq. (6.4) states that each intermediate configuration must satisfy the current GPU resource demand, \( g(c_0) \).

Optimization 2:

Minimize \( \sum_{a_i \epsilon A_n} d(a_i) \) \hspace{1cm} (6.5)

Subject to Constraints:

for all \( c_i \epsilon [c_1, c_2, ..., c_n] \) \hspace{1cm} (6.6)

\[ p(c_i) < P \] \hspace{1cm} (6.7)

\[ g(c_i) = g(c_0) \] \hspace{1cm} (6.8)

Let \( A_n = a_0, a_1, ..., a_{n-1} \) be the sequence of adaptation actions required to transform the node configuration from \( c_0 \) to \( c_n \). Let \( d(a_i) \) denote the length of each adaptation action. The objective Eq. (6.5) is to find the optimal sequence of adaptation actions that minimizes the time required to reach one of the target configurations. We represent the optimization problem as a single source shortest path problem of graph theory. Here, each node configuration is a vertex in the graph and the adaptation actions become the edges between various vertices. The edges are weighted by the adaptation delay, \( d(a_i) \). pVOCL power optimizer applies the Dijkstra’s algorithm to solve the optimization problem.
6.3.3.3 Migration Manager

Figure 6.11 shows that based on the optimal configuration suggested by the power optimizer, the migration manager performs node reconfiguration as follows.

1. Check whether the target configuration requires turning on new nodes.

2. Turn on the required number of nodes at various power-phases of the datacenter cabinets according to the target configuration. Telnet interfaces to various CDUs are used to perform this operation.

3. Based on the VGPU mapping information of the existing configuration and that of the target configuration, identify the VGPUs that need to be migrated from one proxy node to another.

4. Call the `MPI_Comm_connect()` function to establish communication channel with each source proxy node.

5. Call `MPI_Isend()` with the MPI message tag `MAP_MIGRATION` along with the required information to trigger VGPU migration from the source proxy nodes to the destination proxy nodes.

6. Turn off the nodes that do not have any VGPU mapping in the target configuration. Telnet interfaces to the corresponding CDUs are used for this purpose.
Table 1: The workload mix.

<table>
<thead>
<tr>
<th>Node 1</th>
<th>Kernel</th>
<th>Input Size</th>
<th>Kernel instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node 1 GPU 1</td>
<td>N-body</td>
<td>15360</td>
<td>20</td>
</tr>
<tr>
<td>Node 1 GPU 2</td>
<td>Matrix Multiplication</td>
<td>4kx4k</td>
<td>60</td>
</tr>
<tr>
<td>Node 2 GPU 1</td>
<td>Matrix Transpose</td>
<td>3kx3k</td>
<td>20</td>
</tr>
<tr>
<td>Node 2 GPU 2</td>
<td>Matrix Multiplication</td>
<td>3kx3k</td>
<td>60</td>
</tr>
<tr>
<td>Node 3 GPU 1</td>
<td>N-body</td>
<td>23040</td>
<td>20</td>
</tr>
<tr>
<td>Node 3 GPU 2</td>
<td>Matrix Transpose</td>
<td>4kx4k</td>
<td>60</td>
</tr>
</tbody>
</table>

(a) the workload.  
(b) power consumption.

Figure 6.12: Power consumption comparison with different power management techniques.

6.4 Evaluation

We evaluate the pVOCL framework in a testbed of GPU-enabled server cluster. Each computing node is equipped with Dual Intel Xeon Quad Core CPUs, 16 GB of memory, and two NVIDIA Tesla C1060 GPUs. They are installed with the Ubuntu Linux operating system and the CUDA 4.2 toolkit. The servers residing on datacenter cabinets are powered by the switched CW-24VD/VY 3-Phase CDU (Cabinet Power Distribution Unit). The CDUs provide power distribution and remote power monitoring capability. We use the MPICH2 MPI implementation for the compute nodes, which are connected with Ethernet.

6.4.1 Impact of GPU Consolidation

We now demonstrate the impact of GPU consolidation performed by pVOCL on the power consumption, energy usage and application performance. In this experiment, we use three GPU-enabled compute nodes located in the same power-phase circuit of a CDU. Power consumption is measured at time intervals of one second, in the particular power-phase circuit only. Table 1 shows the initial placement of a workload mix.
We compare the instantaneous power consumption of the system under the influence of three power management techniques. First, the hardware technique (h/w-pm) refers to the in-built idle power management of GPU hardware that causes GPUs to save power when they are idle. Second, the hardware/software static power management (h/w-s/w static-pm) is a combination of h/w-pm and a software based technique that turns off a compute node when all its GPU resources are idle. Finally, pVOCL essentially combines GPU consolidation with the first two techniques. In this experiment, pVOCL is configured to find optimal configurations within the same power-phase circuit since we are measuring the impact of GPU consolidation alone without power-phase awareness.

As shown in Figure 6.12(b), pVOCL reduces the power consumption more significantly than the other two techniques. It is due to the fact that pVOCL drives the system towards more power-efficient configurations by pro-actively consolidating GPUs into fewer compute nodes and turning off the unused nodes at time 63 sec and 137 sec. On the other hand, the h/w-s/w static-pm technique waits until time 137 sec to turn off node 2 when all of its GPUs become idle. Then, it turns off node 1 at time 213 sec. The relatively high power consumption in case of h/w-pm technique is due to the fact that the compute nodes are never turned off.

The application performance under the influence of pVOCL is similar to that of the other two techniques. It is due to the fact that pVOCL is able to consolidate GPUs by performing VGPU migration with negligible performance overhead. We measure the overall energy usage of the system under the influence of the three power management techniques. pVOCL improves the energy efficiency by 43% compared to the h/w-pm technique.
Next, we study the impact of workload mix variations on the energy efficiency of the three power management techniques as shown in Figures 6.13 (a) and 6.13 (b). We measure the total energy usage of the system for a fixed period of time using eight variations of the workload mix shown in Table 3.2. In each variation, we change the number of kernel instances in such a way that the average program execution time remains unchanged while the standard deviation varies. In case of h/w-s/w static-pm technique, a higher deviation of the program execution time results in longer waiting times to turn off a node until all of its GPUs become idle. Hence, the total energy usage increases with increase in the deviation of execution time. For the h/w-pm technique, the energy usage remains unchanged since the average program execution time is the same for all workload variations. Similarly, the energy usage for pVOCL is largely unaffected by the increase in the deviation of execution time for the workload mixes used in this experiment. As shown in Figure 6.13 (b), the improvement in energy efficiency by pVOCL reaches up to 29% compared to the h/w-s/w static-pm technique due to variations in the workload mix.

6.4.2 Power-Phase Topology Aware GPU Consolidation and Placement

We evaluate the effectiveness of pVOCL's power-phase topology aware GPU consolidation and placement. We use three compute nodes in each power-phase circuit of the 3-phase CDU. Since each compute node has two GPUs, there are six GPUs in each power-phase. As a case study, we execute N-body kernel benchmarks with input sizes of 15360, 23040, 30720, 38400, 46080, and 53760 bodies respectively on the six GPUs. The number of kernel instances being executed are 20, 40 and 80 respectively on the GPUs residing in the three power-phase circuits. Figure 6.14(a) shows the time-varying demand for GPU resources in the three power-phases as various application kernels finish execution. We measure the power consumption of the entire CDU at time intervals of one second. The peak power budget of the CDU is set to be 4800 Watts in this experiment.

Figures 6.14(b) and 6.14(c) compare the dynamic distribution of GPU workloads across the three power-phases due to pVOCL’s power-phase topology aware consolidation and a consolidation approach without power-phase awareness. Figures 6.15(b) and 6.15(c) show the number of compute nodes that are powered on change accordingly. In case of power-phase aware consolidation, the number of busy GPUs and the
powered-on compute nodes are more evenly distributed across the three power-phases. As a result, as shown in Figure 6.15(a), it is able to maintain a much lower power-phase imbalance in the face of time-varying GPU resource demand. Note that there is a gradual increase in power-phase imbalance in both cases of consolidation. It is due to the fact that there are a lower number of GPU workloads available to balance the power-phases as the benchmark applications finish their execution.

Figure 6.16 shows the impact of power-phase awareness on the instantaneous power consumption of the system. Overall, there is an improvement of 14% in energy efficiency due to the power-phase aware GPU consolidation that drives the system towards more power-efficient configurations with negligible performance overheads. We observed similar results with various other application benchmarks, which are omitted here due to space constraints.

6.4.3 Peak Power Management and Energy Efficiency

We evaluate pVOCL’s ability to manage the peak power consumption while improving the energy efficiency of GPU-enabled server clusters. In this experiment, we start with three compute nodes in one of the power-
Figure 6.16: Impact of power-phase topology aware consolidation and node placement on power consumption.

Figures 6.17(a), (b) and (c) compare the instantaneous power consumption of the entire CDU under the influence of pVOCL’s GPU consolidation and placement actions, when different power budgets are imposed. In each case, pVOCL is able to keep the peak power consumption below the given power budget. However, the decrease in power consumption due to pVOCL occurs much earlier when the power budget is higher. It is due to the fact that pVOCL performs different adaptation actions under different power constraints as shown in Figures 6.18 and 6.19. A higher power budget provides more node reconfiguration options. Hence, pVOCL is able to find the best sequence of adaptation actions to reach a power-efficient configuration under the given power constraint.

Figure 6.18(a) shows that pVOCL initially turns on two compute nodes in power-phase 2 and one compute node in power-phase 1 without violating the power budget of 2600 Watts. This explains the initial increase in power consumption shown in Figure 6.17(a). On the other hand, When the power budget is 2300 Watts, it can turn on only one compute node in power-phase 2 initially as shown in Figure 6.17(b). When the power budget is only 2000 Watts, it is not able to turn on any new compute node until one of the GPUs become idle around time 134 seconds. As a result, a higher power budget allows pVOCL to distribute the GPU workloads across different power-phases and reach more power-efficient configurations much earlier compared to the cases when the power budget is low. This is illustrated in Figures 6.19(a), (b) and (c).

Figures 6.20(a), (b) and (c) show the performance overhead caused by pVOCL’s adaptation actions under various power constraints. We measure the total execution time of each application under the influence of
(a) with power budget 2600 W.  (b) with power budget 2300 W.  (c) with power budget 2000 W.

Figure 6.17: Power consumption trends under various peak power constraints.

(a) with power budget 2600 W.  (b) with power budget 2300 W.  (c) with power budget 2000 W.

Figure 6.18: Node configurations under various peak power constraints.

(a) with power budget 2600 W.  (b) with power budget 2300 W.  (c) with power budget 2000 W.

Figure 6.19: GPU workload distribution under various peak power constraints.
pVOCL and compare it with its default execution time. Note that the applications may undergo different number of migrations or may not be migrated at all due to various GPU consolidation actions taken by pVOCL. This explains the variations in the performance overhead for different applications and power constraints. Furthermore, the performance overhead also depends on the input size of each application. We observe a negligible performance overhead of less than 0.15% in all three cases. It is due to the fact that the amount of time required to migrate VGPUs among various compute nodes is quite small compared to the computational cost. Furthermore, turning on new compute nodes does not impact application performance since the applications continue to execute in the existing compute nodes until the new nodes become available for migration.

Finally, we compare the energy efficiency of pVOCL with the h/w-pm and h/w-s/w static-pm techniques. As shown in Figure 6.21, the improvement in energy efficiency due to pVOCL increases with increasing power budgets.
6.5 Summary

The challenge of power management in virtualized GPU environments mainly lie in the dynamic scheduling of power-hungry GPU resources in the face of complex power consumption characteristics of the underlying system infrastructure and the ever-changing GPU resource demand from multiple users. pVOCL supports dynamic placement and consolidation of GPU workloads in a power aware manner. It controls the peak power consumption and improves the energy efficiency of the underlying server system by migration of existing virtual GPUs and power-phase-aware placement of GPU-enabled server nodes in datacenter cabinets. It drives the system towards energy-efficient configuration by taking an optimal sequence of adaptation actions. We have implemented pVOCL in a cluster of GPU-enabled server nodes using four application kernels. Experimental results demonstrate that pVOCL achieves significant savings in energy consumption through dynamic consolidation of GPU workloads. It drives the system toward optimal energy-efficient configurations because of its awareness of the power characteristics of the widely used three-phase power supply in datacenter cabinets. Furthermore, it optimizes the power consumption while considering the transient costs incurred by various adaptations.
Chapter 7

Automated Resource Allocation and Configuration of MapReduce Environment in the Cloud

Cloud systems have attracted big data processing by removing the burden of hardware and software setup from end users. Popular web services such as Amazon Elastic MapReduce utilize Hadoop framework, which provides distributed data processing capability to businesses, researchers, data analysts, and developers. However, they expect that end users determine the type and amount of cloud resources in the form of virtual machines (VMs) for reservation as well as the configuration of numerous Hadoop parameters. Such job provisioning decisions require in-depth knowledge of system internals and laborious but often ineffective parameter tuning. As a result, customers may suffer from a lack of performance guarantee and increased cost of leasing the cloud resources.

We propose to enable automated allocation of heterogeneous cloud resource sets and configuration of Hadoop parameters for meeting job completion deadlines while minimizing the incurred cost at the same time. There are significant challenges in achieving the goal. First, the heterogeneity of available resource sets in terms of hardware configurations and different pricing schemes complicate the resource allocation decision for various jobs and different input data sizes. Second, Hadoop has over 180 configuration parameters that have varying impact on job performance. Examples include the number of reducers to be launched, the size of memory buffer to use while sorting map output, the number of concurrent connections a reducer should use when fetching its input from mappers etc. Furthermore, the optimal configuration of Hadoop parameters is tightly coupled with hardware configuration of selected resource sets as well as the type of jobs submitted.
We propose and develop AROMA, a system that automates the allocation of heterogeneous Cloud resources and auto-configuration of Hadoop MapReduce parameters. It can be applied as a Cloud service that manages the provisioning of Hadoop jobs. One challenge in automating the resource allocation and configuration of jobs in Hadoop is a lack of performance model for such a complex distributed processing framework. A recent study focused on intensive profiling of routinely executed jobs in the Hadoop environment in order to estimate their performance for various input data sizes [125]. However, such an approach is not feasible for ad-hoc jobs submitted to the system, which have unpredictable execution characteristics. One possible workaround for ad-hoc jobs is to perform intensive job profiling by running them in a staging cluster that is similar to the target Hadoop cluster in size and configuration. However, there are overheads in terms of resources usage and the time spent in job profiling. Hence, it is not suitable for ad-hoc jobs that have deadlines.

AROMA addresses the challenges of Hadoop job provisioning with a novel two-phase machine learning and optimization framework. It exploits an important observation that jobs with similar resource consumption pattern would face similar bottlenecks and as a result they would exhibit similar performance behavior in relation to the changes in resource allocation and Hadoop configuration. In the first offline phase, it groups the information about past jobs into different clusters using the classic $k$-medoid clustering technique. Each cluster consists of jobs that exhibit similar CPU, network and disk utilization patterns. Then for each cluster, it trains a support vector machine (SVM) model that is able to make accurate and fast prediction of a job’s performance for various combinations of resource allocation, configuration parameters and input data sizes.

In the second online phase, it obtains the resource utilization pattern of a newly submitted job by running it in a staging cluster of small VMs with default configuration parameters and using only a fraction of input data to capture its resource utilization signature. We assume that the entire input data is available, when a new job is submitted. This profiling is a lightweight process both in terms of resource usage and the time spent in capturing the signature. AROMA matches the resource utilization signature to those job clusters and identifies the performance model to use for finding the best configuration and resource allocation. Finally, AROMA applies a pattern search based optimization technique on the selected job cluster’s performance model to find close to optimal resource allocation and configuration parameters that meet a Hadoop job’s
completion deadline at the minimal resource cost.

7.1 Motivational Case Study

An interesting artifact of the utility nature of the cloud paradigm is that the cost of using 1000 virtual machines (VMs) for one hour is the same as using one virtual machine (VM) of the same capacity for 1000 hours. Thus, a MapReduce job can potentially improve its performance while incurring the same currency cost by acquiring several machines and executing in parallel. However, a user submitting jobs to a Cloud service has to choose from a variety of virtual machines, which have different hardware configurations and corresponding pricing policies usually expressed as $perhour. For a given MapReduce job with the certain input data set, it is challenging to determine how many and what type of virtual machines should be allocated to meet the job completion deadline while minimizing the incurred cost.

Furthermore, the configuration parameters of the Hadoop MapReduce framework for each virtual machine can significantly affect the performance of a running job. There is no single MapReduce configuration that can optimize the performance of all types of VMs and all types of jobs. To illustrate this, we conduct a case study in a testbed of virtualized Hadoop nodes based on VMware vSphere 4.1. A MapReduce benchmark job Sort, which does a partial sort of its input, is executed with 20 GB input data on a cluster of six VMs. We measure the job execution time for various combinations of resource type and configuration parameters as shown in Table 3.2. For the sake of clarity we use only one configuration parameter and two types of VMs in this case study. We allocate 1 CPU core with 2 GB RAM to a small VM and 2 CPU cores with 4 GB RAM to a medium VM. The usage costs of small and medium VMs are assumed to be 0.85$perhour and 1.275$perhour, respectively. We assume that the maximum number of parallel mappers and reducers are fixed to 1 slot per node for small VMs and 2 slots per node for medium VMs.

The results in Table 1 illustrate the impact of resource allocation and configuration on both performance and cost of running a Hadoop job. We observe that the performance impact of MapReduce parameters significantly varies with resource allocation. For example when six small VMs are allocated, the optimal configuration is given by case 3 with mapred · reduce · tasks equal to 70. However for six medium VM allocation, the optimal configuration is Case 5 with mapred · reduce · tasks equal to 35. In this particular
Table 1: Cost and performance impact of resource allocation and parameter configuration.

<table>
<thead>
<tr>
<th></th>
<th>Hadoop MapReduce Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>small VMs</td>
<td></td>
</tr>
<tr>
<td>Case 1</td>
<td>mapred.reduce.tasks 7</td>
</tr>
<tr>
<td></td>
<td>Exec.time 17.2min</td>
</tr>
<tr>
<td></td>
<td>Cost (cents) 146.2</td>
</tr>
<tr>
<td>Case 2</td>
<td>mapred.reduce.tasks 35</td>
</tr>
<tr>
<td></td>
<td>Exec.time 14min</td>
</tr>
<tr>
<td></td>
<td>Cost (cents) 119.0</td>
</tr>
<tr>
<td>Case 3</td>
<td>mapred.reduce.tasks 70</td>
</tr>
<tr>
<td></td>
<td>Exec.time 13min</td>
</tr>
<tr>
<td></td>
<td>Cost (cents) 110.5</td>
</tr>
<tr>
<td>medium VMs</td>
<td></td>
</tr>
<tr>
<td>Case 4</td>
<td>mapred.reduce.tasks 7</td>
</tr>
<tr>
<td></td>
<td>Exec.time 11.5min</td>
</tr>
<tr>
<td></td>
<td>Cost (cents) 146.6</td>
</tr>
<tr>
<td>Case 5</td>
<td>mapred.reduce.tasks 35</td>
</tr>
<tr>
<td></td>
<td>Exec.time 7.36min</td>
</tr>
<tr>
<td></td>
<td>Cost (cents) 93.8</td>
</tr>
<tr>
<td>Case 6</td>
<td>mapred.reduce.tasks 70</td>
</tr>
<tr>
<td></td>
<td>Exec.time 8min</td>
</tr>
<tr>
<td></td>
<td>Cost (cents) 102.0</td>
</tr>
</tbody>
</table>

study, we have a counter intuitive observation that it may be cheaper to run a Hadoop job using six medium VMs instead of six small VMs. However, the situation can be further complicated by the fact that cost and performance impact can also vary with the different number of VMs and different input size of jobs running in a Hadoop cluster. Hence, it is important to consider all these factors for making a cost efficient and effective job provisioning decision.

7.2 AROMA: Automated Resource Allocation and Configuration of MapReduce Environment in the Cloud

7.2.1 Architecture and Design

AROMA is an automated job provisioning system that integrates resource allocation and parameter configuration to meet job deadline guarantee and improve the cost efficiency of Hadoop MapReduce in Clouds. Figure 7.1 shows the architecture of AROMA. End users submit jobs as input to AROMA through a command line interface. AROMA first calculates the number and type of VMs to be allocated, and assigns appropriate
MapReduce configuration parameters to meet its completion deadline at minimal cost. Then, its resource allocator powers on a pool of dormant VMs pre-installed with Hadoop software. Finally, the job manager submits the job to the Hadoop master node along with its configuration parameters. Here, the key challenges lie in making fast and effective decisions at job submission time, allocating the right amount of resources and finely tuning the MapReduce configuration settings for an ad-hoc job with unpredictable input data size and completion deadline.

The core components of AROMA work in two phases. First the data collector, clustering and performance modeling components process Hadoop log files and resource utilization data to learn the performance models for various types of Hadoop jobs in the offline phase. Next in the online phase, the job profiler, resource signature matcher and optimizer select the appropriate performance model for an incoming job based on its resource utilization signature and calculate the optimal resource allocation and configuration that meet the job completion deadline.

7.2.2 Machine Learning Performance Modeling

The first challenge for making optimal job provisioning decisions is the lack of the performance models for different Hadoop jobs. AROMA learns the performance models of various types of jobs in the offline phase.

7.2.2.1 Data collection and clustering

The data collector component extracts the execution time, input data size, resource allocation and MapReduce configuration parameters of various past jobs from the Hadoop JobTracker log files. For each job, it fetches the corresponding series of resource utilization data of the slave VM nodes. We use `dstat` tool at each slave VM to measure CPU usage (in terms of User, System, Idle and Wait percentages), disk usage (number of blocks in and out) and network usage (bytes/sec into and out of network card) every second. All the usage measurements are normalized using their respective maximum values. In our testbed, the JobTracker and slave VM logs are saved in a repository when a Hadoop cluster is decommissioned after job completion.

A core principle that enables AROMA’s adaptiveness to ad-hoc jobs is based on the observation that jobs with similar resource utilization pattern exhibit similar performance behavior. As a result, AROMA needs
Figure 7.2: CPU utilization of Sort, Wordcount and Grep jobs during different runs.

(d) Sort run2. (e) Wordcount run2. (f) Grep run2.

Figure 7.3: Euclidean Signature Distances for CPU, Network and Disk resources for (i) Sort vs Sort;(ii) Sort vs Wordcount;(iii) Sort vs Grep;

to learn only a single performance model for a group of similar jobs. The resource utilization signature of each job is a combination of time series data of average CPU usage, average memory usage and average disk usage rate. However, it is difficult to compare the resource signatures of two jobs accurately in a multi-tenant cloud environment. It is due to the fact that the contention of shared resources between multiple applications introduces noise and outliers in the resource utilization patterns.

We examine the variability of resource utilization patterns of various Hadoop jobs running on a small cluster of 2 small VMs and using 1 GB input data. The Hadoop configuration parameters are set to the default value. To simulate a multi-tenant cloud environment, we host another group of VMs in the same
physical server cluster of our testbed and execute Hadoop jobs in them. Figures 7.2 (a) and 7.2(d) compare the CPU utilization patterns of a VM node when a Sort job is executed twice. Similarly, the CPU utilization patterns of Wordcount and Grep jobs are also compared. We observed variations in the utilization patterns of the same job when it is executed at different times.

It is challenging to accurately identify the degree of similarity between different jobs due to the disturbance in the resource utilization pattern of VMs. We demonstrate this challenge by measuring the difference between the utilization patterns of various job combinations using a straightforward Euclidean distance metric. Figure 7.3 shows the Euclidean distance between the resource utilization signatures of different job combinations. We observe that the Euclidean distance based resource utilization signature comparison is misleading. For instance according to this metric, the signature distance of CPU and network utilization between two executions of the same Sort job is found to be greater than that of Sort vs Wordcount job executions.

AROMA addresses this challenge by applying a Longest Common Subsequence (LCSS) based distance metric. Unlike a commonly used Euclidean Distance metric, LCSS is robust against noise and outliers in a resource utilization pattern. It is also effective in comparing the similarity between patterns that are out of phase and of different lengths. Figure (7.4) shows LCSS distance between the resource utilization signatures of different job combinations. The distances between the CPU, disk and network utilization patterns of two Sort jobs is the smallest and the distances between the utilization patterns of Sort and Grep jobs are the largest.

AROMA applies the classic k-medoid based clustering technique along with LCSS distance metric to group jobs with similar utilization patterns of CPU, network and disk resources. According to the resource
Table 2: Feature Selection for SORT performance model.

<table>
<thead>
<tr>
<th>Features</th>
<th>t-statistic</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>io.sort.factor</td>
<td>-3.6306</td>
<td>0.0211</td>
</tr>
<tr>
<td>mapred.job.shuffle.merge.percent</td>
<td>0.9613</td>
<td>0.3423</td>
</tr>
<tr>
<td>io.sort.spill.percent</td>
<td>2.13</td>
<td>0.2014</td>
</tr>
<tr>
<td>mapred.job.shuffle.input.buffer.percent</td>
<td>-1.5352</td>
<td>0.1328</td>
</tr>
<tr>
<td>io.sort.record.percent</td>
<td>-1.0588</td>
<td>0.0191</td>
</tr>
<tr>
<td>mapred.job.reduce.input.buffer.percent</td>
<td>1.4043</td>
<td>0.1682</td>
</tr>
<tr>
<td>mapred.reduce.parallel.copies</td>
<td>4.3869</td>
<td>0.0342</td>
</tr>
<tr>
<td>io.sort.mb</td>
<td>-3.3465</td>
<td>0.0312</td>
</tr>
<tr>
<td>mapred.reduce.tasks</td>
<td>-6.4583</td>
<td>0.0101</td>
</tr>
<tr>
<td>input size</td>
<td>10.8</td>
<td>0.0011</td>
</tr>
<tr>
<td>VM type</td>
<td>8.5</td>
<td>0.0045</td>
</tr>
<tr>
<td>number of VMs</td>
<td>-4.2</td>
<td>0.0021</td>
</tr>
</tbody>
</table>

Table 3: Feature Selection for GREP performance model.

<table>
<thead>
<tr>
<th>Features</th>
<th>t-statistic</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>io.sort.factor</td>
<td>-1.1324</td>
<td>0.2682</td>
</tr>
<tr>
<td>mapred.job.shuffle.merge.percent</td>
<td>-1.4953</td>
<td>0.1474</td>
</tr>
<tr>
<td>io.sort.spill.percent</td>
<td>-1.4527</td>
<td>0.1587</td>
</tr>
<tr>
<td>mapred.job.shuffle.input.buffer.percent</td>
<td>-0.8069</td>
<td>0.4273</td>
</tr>
<tr>
<td>io.sort.record.percent</td>
<td>3.8211</td>
<td>0.0007</td>
</tr>
<tr>
<td>mapred.job.reduce.input.buffer.percent</td>
<td>3.8016</td>
<td>0.0008</td>
</tr>
<tr>
<td>mapred.reduce.parallel.copies</td>
<td>-0.1715</td>
<td>0.8652</td>
</tr>
<tr>
<td>io.sort.mb</td>
<td>9.487</td>
<td>0.0112</td>
</tr>
<tr>
<td>mapred.reduce.tasks</td>
<td>-0.7814</td>
<td>0.4419</td>
</tr>
<tr>
<td>input size</td>
<td>9.3</td>
<td>0.0033</td>
</tr>
<tr>
<td>VM type</td>
<td>7.5</td>
<td>0.0056</td>
</tr>
<tr>
<td>number of VMs</td>
<td>-4.5</td>
<td>0.0011</td>
</tr>
</tbody>
</table>

utilization patterns of the three jobs, AROMA groups Sort and Wordcount jobs in the same cluster and the Grep job belongs to a different cluster. Resource signatures implicitly capture the potential impact of input data content on a job’s behavior.

7.2.2.2 Performance modeling

AROMA applies a powerful supervised machine learning technique to learn the performance model for each cluster of jobs. It constructs a support vector machine (SVM) regression model to estimate the completion time of jobs belonging to a cluster for different input data sizes, resource allocations and configuration parameters. SVM methodology is known to be robust for estimating real-valued functions (regression problem) from noisy and sparse training data having many attributes. This property of SVM makes it a suitable
technique for performance modeling of complex Hadoop jobs in the Cloud environment.

It is important to create a model that utilizes only the statistically significant features and avoids “overfitting” the data. For each job cluster, we apply a stepwise regression technique to determine which set of features are the best predictors for the job performance. It is a systematic method for adding and removing terms from a model based on their statistical significance in a regression. The method begins with an initial model and then compares the explanatory power of incrementally larger and smaller models. At each step, the $p$ value of a $t$-statistic is computed to test models with and without a potential term. If a term is not currently in the model, the null hypothesis is that the term would have a zero coefficient if added to the model. If there is sufficient evidence to reject the null hypothesis, the term is added to the model. Conversely, if a term is currently in the model, the null hypothesis is that the term has a zero coefficient. If there is insufficient evidence to reject the null hypothesis, the term is removed from the model.

We conduct stepwise regression on the data sets collected from our testbed of virtualized Hadoop nodes. For data collection, we measured the execution times of various Hadoop jobs with different input data sizes in the range 5 GB to 50 GB, using different Hadoop configuration parameters and running on different cluster sizes of Hadoop nodes comprising of small and medium VMs. Tables 2 and 3 show the results of stepwise regression for two different Hadoop jobs, Sort and Grep respectively. Note that the features whose $p$-values are smaller than or equal to the significance level of 0.05 are selected for performance modeling using SVM regression.

The idea of SVM regression is based on the computation of a linear regression function in a high dimensional feature space where the input data are mapped via a nonlinear function. The linear model in an $M$ dimensional feature space $f(x, \omega)$ is given by

$$f(x, \omega) = \sum_{m=1}^{M} \omega_m g_m(x) + b$$  \hspace{1cm} (7.1)

where $g_m(x)$ denotes a set of nonlinear transformations and $b$ is the bias term. The input data $x$ for AROMA’s performance model consists of the features selected by the stepwise regression method.

AROMA applies $\varepsilon - SV$ regression technique [17] that aims to find a function $f(x, \omega)$ that has at most $\varepsilon$ deviation from the actual output values $y$ for all the training data points. In this case, the output values of the performance model are the job execution times. At the same time, it also tries to reduce model complexity
by minimizing the term $||\omega||^2$. We apply the LIBSVM [17] library to find suitable kernel functions and train the SVM regression model.

SVM regression has a good generalization and prediction power due to the fact that unlike other learning techniques, it attempts to minimize a generalized error bound so as to achieve generalized performance. This error bound is the combination of the training error and a regularization term that controls the complexity of the model. One approach for assessing how accurately a predictive model will perform in practice is to perform *cross-validation*. We apply leave-one-out cross-validation (LOOCV), which involves using a single observation from the original data sample as the validation data, and the remaining observations as the training data. This is repeated such that each observation in the sample is used once as the validation data. Figure 7.5 shows the results of cross validation performed for three Hadoop jobs in terms of the root mean square error in prediction. AROMA’s SVM regression based performance model is able to achieve significantly better prediction accuracy than a linear regression model.

### 7.2.3 Online Job Profiling and Signature Matching

A MapReduce environment needs to run ad-hoc jobs with unpredictable resource utilization patterns that make it difficult to apply performance models. AROMA addresses this challenge by first running a new job in a small set of Hadoop nodes and using a small chunk of input data. It collects the resource utilization signature of the job using *dstat* tool. Then the signature is compared with the resource utilization signatures of various job clusters’ centroid. We observe that signature matching with the cluster centroid rather than each data point is sufficient since the data points within the same cluster are similar. As a result, this process
can be executed with less overhead.

However, a straightforward comparison of resource utilization signatures may cause misleading results due to the presence of noise and outliers. AROMA applies the LCSS distance metric to match the signatures corresponding to each resource type for a sequence of measurement samples. It is able to make accurate comparison between the resource utilization patterns of two jobs.

### 7.2.4 Cost Efficient Performance Guarantee

It is important but challenging to make effective job provisioning decisions to meet performance guarantee in terms of job completion deadlines and to minimize the cost of resource allocation. AROMA's job provisioning decision is based on a non-linear constrained optimization problem formulated for a given job with input size \( s \) and completion deadline \( D \) as follows:

\[
\text{Minimize } \sum_{i=1}^{N} n_i \cdot rate_i \cdot t_s
\]

Subject to Constraints:

\[
t_s \leq D
\]

\[
\forall j \in [1, C], \quad lb_j \leq conf_j \leq ub_j
\]

Eq. (7.2) gives the optimization objective to minimize the total cost of allocating VMs to the Hadoop cluster. Here, \( n_i \) is the number of VMs of type \( i \) (small,medium) that is charged a cost of \( rate_i \) at $/hour and \( t_s \) is the execution time of the given job with input data size \( s \). Eq. (7.3) defines a constraint that the job execution time must be less than its completion deadline \( D \). Eq. (7.4) defines the lower bound \( lb \) and upper bound \( ub \) values of each MapReduce configuration parameter under consideration.

In this optimization problem, the relationship between the decision variables \((n_i, con_{f_j})\) and the dependent variable \( t_s \) is given by the SVM regression model corresponding to a given job type. We solve the optimization problem by applying a pattern search algorithm, the generating set search. It is a direct search method that can optimize complex objective functions that are not differentiable.

It computes a sequence of points that approach an optimal point. At each step, the algorithm searches a set of points, called a mesh, around the current point computed at the previous step of the algorithm. The mesh is formed by adding the current point to a scalar multiple of a set of vectors called a pattern. If the
pattern search algorithm finds a point in the mesh that improves the objective function at the current point, the new point becomes the current point at the next step of the algorithm.

### 7.3 Implementation

We implement AROMA on a testbed consisting of seven HP ProLiant BL460C G6 blade server modules and a HP EVA storage area network with 10 Gbps Ethernet and 8 Gbps Fibre/iSCSI dual channels. Each blade server is equipped with Intel Xeon E5530 2.4 GHz quad-core processor and 32 GB PC3 memory. Virtualization of the cluster is enabled by VMware ESX 4.1. We create a pool of VMs with different hardware configurations from the virtualized blade server cluster and run them as Hadoop nodes. There are small VMs with 1 vCPU, 2 GB RAM and 50 GB hard disk space. Medium VMs have 2 vCPUs, 4 GB RAM and 80 GB hard disk space. Each VM uses Ubuntu Linux version 10.04 and Hadoop 0.20.2.

We designate one blade server to host the master VM node and use rest of the servers to host the slave VM nodes. The single master node runs the JobTracker and the NameNode, while each slave node runs both the TaskTracker and the DataNode. A small slave VM is configured with one Map slot, one Reduce slot and 200 MB memory per task. Whereas, a medium type slave VM is configured with two Map slots, two Reduce slots and 300 MB memory per task. The data block size is set to 64 MB. The AROMA job provisioning system can be run either on a separate VM or a standalone machine as it manages the Hadoop nodes remotely.

As related studies [59, 125], we use a number of benchmark jobs that come with the Hadoop distribution for performance evaluation, i.e., Sort, Grep, and Wordcount. We use the RandomWriter and RandomTextWriter tools in the Hadoop package to generate data of various sizes for the Sort, WordCount and Grep programs.

### 7.4 Evaluation

We evaluate AROMA’s capability for predicting the performance of Hadoop jobs, auto-configuration of Hadoop MapReduce parameters, joint resource allocation with configuration, and adaptiveness to ad-hoc Hadoop jobs for improving job performance and reducing the incurred cost. For our experiments, the costs
of using a small VM and a medium VM are assumed to be $0.85 per hour and $1.275 per hour respectively.

### 7.4.1 AROMA Performance Model Accuracy

First, we evaluate AROMA’s ability to predict the completion times of Hadoop jobs when they are executed with different number of VM nodes of different types and different input data sizes. We use the default Hadoop configuration for this experiment. Figures 7.6 (a), (b) and (c) show the actual and predicted running
times of Sort, Wordcount and Grep jobs respectively as the number of Hadoop slave nodes are varied. In this case, we use small VMs as Hadoop nodes and fix the input data to be 20 GB. Figures 7.7 (a), (b) and (c) show the results of a similar study using medium VMs. We observe that AROMA’s SVM regression based performance model is able to predict the speedup achieved for each job as we increase the number of VMs. The relative error between the actual and predicted job completion time is less than 12% on all the cases. We observed similar prediction accuracy when the input data size is varied while keeping the number of VMs fixed as shown in Figure 7.8. Here, we use six small VMs to run a Sort job.

7.4.2 Auto-Configuration

We evaluate AROMA’s capability to automatically configure the Hadoop environment parameters according to variations in the resource allocation. We execute the Hadoop Sort benchmark. The jobs are executed for an input data of 10 GB. Figures 7.9 (a), (b) and (c) compare the impact of using the default Hadoop configuration with AROMA’s auto-tuned configuration on the job execution time and the incurred cost. AROMA’s auto-
configuration significantly outperforms the default configuration for various allocations of small VMs. The improvement in job performance and cost due to AROMA increases from 17% to 30% when using more number of VMs. It is due to the fact that there are more opportunities for configuration tuning in case of using more VMs. Figures 7.10 (a), (b) and (c) show the job execution results when medium VMs are used. We observe that using more than six medium VMs shows little improvement in performance and hence increases the cost. It is due to the fact that for a moderately small input data of 10 GB, the overhead of managing a large number of Map/Reduce slots in the cluster can be significant.

Note that in Figure 7.9(c) and Figure 7.10(c), the improvement percentage in performance and in cost are the same since the cost of using a VM depends on how long a job runs.

Table 4 compares the Hadoop configurable parameter values due to AROMA’s auto-configuration with the default Hadoop values. AROMA uses this configuration for running Sort benchmark with six small VMs and input data of 20 GB. We can observe that there are many differences between two settings. In particular, parameters `io.sort.factor`, `mapred.reduce.tasks`, and `mapred.reduce.parallelcopies` have significantly different values. Hadoop Sort benchmark sets the default value of `mapred.reduce.tasks` to 0.9 times the total number of reduce slots in the cluster.

Please note that for the Sort job even if an end user is able to come up with the same parameter configuration as that of AROMA based on best practices, such a parameter configuration is not effective when the Hadoop environment and the submitted jobs change dynamically. For instance, Table 5 shows the parameter configuration differences for running the Sort job with six medium VM and input data of 20 GB. It shows there are a few important differences between two parameter configurations by AROMA.
Table 4: Hadoop configuration parameters for Sort benchmark (six small VMs and 20 GB input data).

<table>
<thead>
<tr>
<th>Hadoop parameters</th>
<th>AROMA</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>io.sort.factor</td>
<td>300</td>
<td>10</td>
</tr>
<tr>
<td>mapred.job.shuffle.merge.percent</td>
<td>0.66</td>
<td>0.66</td>
</tr>
<tr>
<td>io.sort.spill.percent</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>mapred.job.shuffle.input.buffer.percent</td>
<td>0.7</td>
<td>0.7</td>
</tr>
<tr>
<td>io.sort.record.percent</td>
<td>0.15</td>
<td>0.05</td>
</tr>
<tr>
<td>mapred.job.reduce.input.buffer.percent</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mapred.reduce.parallel.copies</td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td>io.sort.mb</td>
<td>130</td>
<td>100</td>
</tr>
<tr>
<td>mapred.reduce.tasks</td>
<td>120</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 5: Hadoop configuration parameters for Sort benchmark (six medium VMs and 20 GB input data).

<table>
<thead>
<tr>
<th>Hadoop parameters</th>
<th>AROMA</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>io.sort.factor</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>mapred.job.shuffle.merge.percent</td>
<td>0.66</td>
<td>0.66</td>
</tr>
<tr>
<td>io.sort.spill.percent</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>mapred.job.shuffle.input.buffer.percent</td>
<td>0.7</td>
<td>0.7</td>
</tr>
<tr>
<td>io.sort.record.percent</td>
<td>0.3</td>
<td>0.05</td>
</tr>
<tr>
<td>mapred.job.reduce.input.buffer.percent</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mapred.reduce.parallel.copies</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>io.sort.mb</td>
<td>160</td>
<td>100</td>
</tr>
<tr>
<td>mapred.reduce.tasks</td>
<td>120</td>
<td>10</td>
</tr>
</tbody>
</table>

Next, we evaluate the auto-configuration capability of AROMA when the input data size is varied. Figures 7.11(a),(b) and (c) show the performance improvement due to AROMA for running the Sort benchmark by using six small VMs.

### 7.4.3 Efficient Resource Allocation and Configuration

A key feature of AROMA is its holistic job provisioning approach with optimization for joint resource allocation and auto-configuration. We demonstrate the merit of this feature by comparing the cost efficiency in achieving the job completion deadline by AROMA with and without the integration of resource allocation with auto-configuration of Hadoop parameters.

Tables 6 and 7 show the execution time and cost of completing a Hadoop Sort job for various input data sizes by AROMA without and with the integration of resource allocation with auto-configuration respectively. Results show that both approaches can complete the job within the completion deadline 360 seconds. However, AROMA with auto-configuration is much more cost efficient for meeting the job completion deadline.
for all different input data sizes. For example, for the Sort benchmark job with data size 10 GB, AROMA without auto-configuration costs 51 cents to finish the job within the given deadline using 6 medium VMs. On the other hand, AROMA with auto-configuration uses only 5 medium VMs to meet the deadline while incurring the total cost of 36 cents.

The main reason for the cost efficiency of AROMA is two-fold. First, AROMA is able to fine tune the Hadoop configuration parameters corresponding to various resource allocations. As a result, a job running with AROMA’s optimized configuration can finish faster than a job using the default Hadoop configuration for the same set of resource allocations. We have observed similar performance improvement results for other job completion deadlines but omitted here due to the space limit.

Second, it performs a cost-aware optimization for the allocation of heterogeneous resources. As shown in Table 7, when the input data is 5 GB, AROMA allocates three small VMs that complete the job in 210 seconds. We note that using two small VMs could still be able to meet the job completion deadline that is 360 seconds. This kind of counter intuitive behavior of AROMA is in fact advantageous. This is due to the fact that running three small VMs for 210 seconds is cheaper than running two VMs of the same capacity for a much longer time, while still meeting the job completion deadline. The cost saving is 12% in this case. On average, AROMA shows cost efficiency of 25%.

Table 6: Cost of meeting job execution deadline (360 sec).

<table>
<thead>
<tr>
<th>Data input</th>
<th>VMs (number and type)</th>
<th>Execution time (sec)</th>
<th>Cost (cents)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 GB</td>
<td>3 small</td>
<td>300</td>
<td>21</td>
</tr>
<tr>
<td>10 GB</td>
<td>6 medium</td>
<td>240</td>
<td>51</td>
</tr>
<tr>
<td>20 GB</td>
<td>12 medium</td>
<td>358</td>
<td>152</td>
</tr>
</tbody>
</table>

Table 7: Cost of meeting job execution deadline (360 sec).

<table>
<thead>
<tr>
<th>Data input</th>
<th>VMs (number and type)</th>
<th>Execution time (sec)</th>
<th>Cost (cents)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 GB</td>
<td>3 small</td>
<td>210</td>
<td>14.8</td>
</tr>
<tr>
<td>10 GB</td>
<td>5 medium</td>
<td>205</td>
<td>36</td>
</tr>
<tr>
<td>20 GB</td>
<td>10 medium</td>
<td>357</td>
<td>126</td>
</tr>
</tbody>
</table>
7.4.4 Adaptiveness to Ad-hoc Jobs

We evaluate AROMA’s ability to predict the performance of ad-hoc jobs. Assuming that an ad-hoc Wordcount job with 10 GB input data is submitted to the system, AROMA first captures the resource utilization signature of the job by executing it in a staging cluster of two small VMs using 1 GB input data and default Hadoop configuration. As discussed in Section 7.2.2.1, the resource utilization signature of Wordcount job is found to be similar to that of a Sort job. We assume that the performance model of Sort job is available due to offline data clustering and modeling. AROMA calculates the difference between the measured execution time of Wordcount in the staging cluster and the execution time predicted by the SVM performance model of the Sort job. It applies this difference value to scale the predictions made by the performance model for different resource allocations and configurations.

The accuracy of AROMA’s resource signature matching and the performance of ad-hoc jobs depend on the diversity of existing job clusters. Job clustering is initially performed offline based on past data. It can be repeated at regular time intervals to accommodate diverse new jobs.

Figure 7.12 (a) demonstrates the accuracy of AROMA’s performance prediction of Wordcount job for various VM resource allocations. In this case, we use small VMs with default Hadoop configuration for performance evaluation. AROMA is able to achieve a very accurate prediction of job execution time with a relative error of less than 12.5%. Next, we compare the actual and predicted execution times of Wordcount job running on 4 small VMs with 20 various Hadoop configurations as shown in Figure 7.12 (b).
7.5 Summary

The software framework MapReduce and its open-source implementation Hadoop are increasingly deployed in the Cloud to harness the unlimited availability of virtualized resources and pay-per-usage cost model of cloud computing. However, currently end users have to make job provisioning decisions manually at the Hadoop environment using best practices. They suffer from a lack of performance guarantee and increased cost of leasing the Cloud resources.

AROMA is proposed and developed to enable automated allocation of heterogeneous cloud resource sets and configuration of Hadoop parameters for meeting job completion deadlines while minimizing the incurred cost. It addresses the significant challenges of automated and efficient Hadoop job provisioning with a novel two-phase machine learning and optimization framework. It is able to make optimal job provisioning decisions with respect to resource allocation efficiency in the face of unpredictable input data sizes and performance expectations. The significance also lies in the fact that it enables automated job provisioning of Hadoop MapReduce framework in the Cloud so that end users can utilize cloud services without acquiring in-depth knowledge about the system internals or going through laborious and time consuming manual but ineffective configuration tuning.

AROMA is developed upon the popular Hadoop MapReduce environment. It is ready to be tailored for other MapReduce running environments. In the future work, we will extend AROMA for running multi-stage job workflows.
Chapter 8

Conclusions and Future Work

We state the conclusions of this thesis before discussing opportunities for further investigation.

8.1 Conclusions

Modern datacenters have become the platform for supporting cloud computing services. Virtualization is a key technology that has enabled this transition by allowing diverse applications to flexibly share underlying server resources. Due to the highly dynamic nature of Internet workloads, increasing complexity of Internet applications, multi-tier/multi-service architectures and complex dynamics of underlying shared infrastructure, today datacenters face significant and multi-facet challenges in meeting service level agreements with their clients while maintaining resource utilization efficiency and reducing power consumption costs. In this context, there are growing research interests in realizing the goal of self-managing autonomic systems that adapt to dynamic execution environments while hiding system complexity to operators and users.

We have developed middleware approaches to autonomic performance and power control in virtualized datacenters. Firstly, we designed self-adaptive and efficient virtual server provisioning techniques to guarantee a high percentile performance of multi-tier web applications in the face of highly dynamic workloads. Secondly, we proposed and developed a non-invasive and energy efficient mechanism to achieve performance isolation of heterogeneous applications in a virtualized datacenter. Thirdly, we designed a system for coor-
ordinated control of application performance and power consumption in virtualized servers. Furthermore, we
developed a power-aware framework for managing scientific workloads running in GPU clusters, with the
help of emerging technologies that provide GPU accelerators as virtualized computing resources in High Per-
formance Computing datacenters. Finally, we developed an automation tool for effective and cost-efficient
resource allocation and configuration of Hadoop, a popular distributed data processing framework in the
Cloud. We conducted extensive evaluation of the proposed resource management techniques through sim-
ulations and testbed implementation. The evaluation results demonstrate its effectiveness in controlling the
quality of service provided by virtualized resources, and improving the energy efficiency of the underlying
system.

The main technical novelty and contribution of this thesis are due to the integration of the strengths
of machine learning, optimization, and control theoretical techniques to develop self-managing computing
systems that are capable of controlling both power consumption and application performance in a virtualized
datacenter, while reducing the burden of complex system management from human operators.

8.2 Future Work

Computing systems have reached a level of complexity where the human effort required to get the systems
up and running and keeping them operational is getting out of hand. My research will continue to pursue
the coveted vision of autonomic computing, which seeks to improve computing systems while decreasing
human involvement. It aims to expand its scope and deepen the investigation to build self-configuring, self-
optimizing and self-healing systems in the areas of Cloud Computing, Sustainable Computing and Big data
Processing. One of my research thrusts will focus on improving the performance and energy efficiency of
Big Data Processing framework, MapReduce, deployed in the Cloud. I will investigate the impact of resource
heterogeneity and performance variability of cloud systems on application performance, and build autonomic
systems to address such challenges. As cloud systems continue to grow in complexity and scale, reliability
becomes a critical concern. My research will explore automated fault diagnosis, failure prediction, and fault-
tolerance in Cloud computing.
Publications

Journal articles


Conference proceedings


Quality of Service (IWQoS), 10 pages, Montreal, June 2013 (acceptance rate 28%).


8. NINEPIN: Non-Invasive and Energy Efficient Performance Isolation in Virtualized Servers, Palden Lama and Xiaobo Zhou, Proc. of the 42nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 12 pages, Boston, June 2012 (acceptance rate 17%).

9. Automated and Agile Server Parameter Tuning with Learning and Control, Yanfei Guo, Palden Lama and Xiaobo Zhou, Proc. of the 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS), 12 pages, Shanghai, May 2012 (acceptance rate 21%).

10. aMOSS: Automated Multi-Objective Server Provisioning with Stress-Strain Curving, Palden Lama and Xiaobo Zhou, Proc. of the 40th IEEE International Conference on Parallel Processing (ICPP), 10 pages, Taipei, Sep 2011 (acceptance rate 21%).


Submitted (Under peer review)

Bibliography


