1-1-2011

Autonomic management of virtualized resources in cloud computing

Jia Rao
Wayne State University,

AUTONOMIC MANAGEMENT OF VIRTUALIZED RESOURCES IN CLOUD COMPUTING

by

JIA RAO

DISSERTATION

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

2011

MAJOR: COMPUTER ENGINEERING

Approved by:

Advisor Date

ACKNOWLEDGEMENTS

The study for PhD in the past 5 years is intense and full of challenges. I could not have achieved anything without help from many people. I am grateful to my advisor, Dr. Cheng-Zhong Xu, for his invaluable guidance, support, and encouragement throughout this work. I would also like to thank Dr. Gang George Yin, Dr. Leyi Wang, and Dr. Song Jiang for being my committee members and their suggestions to improve this work I am thankful to my friends, Xiangping Bu, Jiayu Gong, Zhen Kong, Kun Wang, Yudi Wei, Yuehai Xu and Xuechen Zhang for their friendship and help.

Finally, I would like to thank my parents for their understanding and unconditional support for my study. I want to give special thanks to my wife, Yao Li, for her love, understanding and support during happy and hard days.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

The last five years have witnessed a rapid growth of cloud computing in business, governmental and educational IT deployment. Hosting services in a cloud gradually supersedes traditional IT products in both small and large companies. It enables small businesses to have on-demand access to resources that would not likely be available if they needed to build the infrastructure themselves; it allows large companies to provide easy and fast application deployment and adaptation to end users, improving hardware utilization in existing infrastructures. The success of cloud services depends critically on the effective management of virtualized resources. In this dissertation work, we aim to design and implement a cloud resource management mechanism that manages underlying complexity, automates resource provisioning and controls client-perceived quality of service (QoS).

In this chapter, we introduce the motivation and background of this dissertation work, discuss the major challenges and present an overview of our solution.

1.1 Motivation and Background

Cloud computing usually refers to anything that involves delivering hosted services over the Internet. The services are made available in a pay-as-you-go manner to general public [6]. One important cloud offering is to deliver computer Infrastructure-as-a-Service (IaaS). In this type of cloud, raw hardware infrastructure, such as CPU, memory, storage and network, is provided to cloud users as an on-demand virtual server or virtual machine (VM). With the illusion of infinite computing resources available on demand, software developers can start with small scale systems and expand hardware resources when application needs increase. It eliminates the up-front commitment by cloud users for over-provisioned hardware resources, which is usually planned for peak application demands.

Aside from client-side reduced total cost of ownership (TCO), a key benefit of IaaS for cloud providers is the increased resource utilization in data centers. Cloud providers can consolidate traditional web applications into fewer physical servers assuming that the peak loads of individual applications have few overlaps with each other. To achieve elasticity at cloud user side and improve resource utilization at cloud provider side, hardware resources need to be multiplexed and shared between different users. This calls for an effective management of virtualized resources that 1) guarantees the Service Level Agreement (SLA) of individual cloud applications in the presence of time-varying application demands and cloud dynamics; 2) improves hardware resource utilization in data centers; 3) works in a real-time manner and provides fine-grained resource control.

Before delving into more details, we define the terms used in this dissertation work:

- **Cloud User** is the one who pays for the use of computing resources from the public clouds on a short-term basis. If cloud users use the leased resources to host services to others, such as an e-commerce website to shoppers, they are also application service providers. To distinguish the users of the hosted services and the users of the IaaS cloud, we term them as *Application Users* and *Cloud Users*, respectively.

- **Cloud Provider** refers to the company that provides cloud-based platform, infrastructure and application services to other organizations and/or individuals, usually with a utility-based payment model. In this dissertation work, we limit the cloud provider to a IaaS cloud.

- **Cloud SLA** should specify the levels of service of the cloud infrastructure, such as availability and performance. Leading cloud providers like Amazon Elastic Compute Cloud (EC2) only define availability objectives for "uptime" but no explicit guarantees on performance, such as the reservation on a specific resource, isolations between users and variations in the resource supply, in the SLA.

- **Cloud Application's SLA** refers to the service contract between the cloud application provider (i.e. cloud user) and the application users (e.g. online shoppers). Like the SLAs provided by traditional IT organizations, a cloud application's SLA also defines service level objectives (SLO) regarding application-specific performance metrics. For example, an e-commerce application may define performance targets for maximum allowable response time and minimum throughput.

In this dissertation work, we focus on the resource management scheme that guarantees individual cloud applications' SLA and treat the cloud infrastructure as a black-box. If not otherwise specified, SLA refers to a cloud application's SLA in the remaining of the work.

1.2 Challenges in Automatic Cloud Resource Management

In this section, we first discuss the challenges in managing the performance of different applications in a shared cloud environment. After that, we discuss the issues need to be addressed in large scale dynamic cloud systems.

1.2.1 Application-centric Performance Management

For the majority of applications deployed on physical hardware, the industry practice was to infer their performance by looking at resource utilizations. As new applications emerge, such as multi-tier websites and web 2.0 applications, it becomes non-trivial to link hardware utilizations to application-level performance. Due to the diversity of applications, it is unlikely appropriate to set a uniform utilization threshold as an indicator of abnormal performance. In a shared hosting environment, in particular a cloud infrastructure, it is desirable to perform application-centric management for individual applications. One challenge is how to manage applications with different performance targets in a unified scale. For example, administrators often need to compare the performance of different applications in order to prioritize applications having larger degradation with more resource allocations. Another challenge is how to combine an application's multiple SLOs into a unified performance index.

1.2.2 Complex Resource to Performance Relationship

In a cloud, application performance depends on the application's ability to simultaneously access multiple type of resources [51]. Accurate resource to performance models are critical to the design of an automatic resource management. However, the workload and cloud dynamics make the determination of a system model challenging.

Time-varying application resource requirement. The intensity and mix of hosted cloud applications can vary considerably over time resulting in changing demands of multiple resources. There are several difficulties in deriving an resource to performance mapping. First, there are inherent non-linear relationship between resource and performance in busy applications making modeling difficult. Second, the interplay of multiple resources, such as CPU, memory and disk further complicates the modeling. Finally, some application demands depend heavily on the inputs, which themselves can not be trivially characterized. For example, an application's memory and disk bandwidth requirements depend on its working set size and I/O request size, both of which can not be easily estimated without intrusive application or guest operating system instrumentation.

Performance interference between co-resident applications. Although server virtualization provides security isolation, fault isolation and environment isolation, it does not guarantee performance isolation between co-resident applications. The interferences may come from centralized virtual machine scheduling. For example, in popular virtualization platforms such as Xen [87] and VMware [76], the processing of privileged instructions, memory writes and I/O requests requires the cooperation of the centralized virtualization layer (i.e. the hypervisor). Thus, the performance of one application may be adversely affected by other applications that aggressively deprive the hypervisor resources. Contentions on shared hardware resources can also cause significant performance variation and degradation to co-running applications. For example, the authors in [96] observed performance degradations as large as 60% in applications sharing the last-level cache (LLC). The performance dependencies on other applications again complicates the resource to performance modeling.

Uncertainties in cloud resources. Although appearing as an infinite and unified resource pool in the front-end, cloud resources are provided by the background multiplexing and virtualization of heterogeneous hardware resources. With identical nominal resource configurations, the actual resources that are available to hosted applications may vary over time and depend on the type of hardware resources behind the cloud. The authors in [22, 78] observed distinct application performance, up to a ratio of 4 on Amazon EC2 VM instances from different service regions.

Commercial virtualization products such as VMware ESX hypervisor try to address resource heterogeneity by allocating resources in terms of some well-recognized metrics. For example, CPU resources are measured by frequency (i.e. MHz) and disk I/O is measured by bandwidth (i.e. MByte per second). However, these metrics can not address all the performance variation due to resource heterogeneity. Processors in new generations may have better pipeline design and larger on-chip cache but are with a lower frequency. Even with the same processor, scheduling can also cause significant performance variations on modern multi-core architectures with heterogeneous cores and hardware hyperthreading. Cloud users may experience different virtual disk performance due to disk technologies like Zone-Bit-Recording (ZBR) that cause different data transfer rates in cloud storage.

1.2.3 Process Delays in Resource Allocation.

Process delay is the time between allocating resources and accurate measuring the effect of the resource allocation on application QoS. In fine-grained cloud management, VM resource allocation relies on precise operations that set resources to desired values assuming the observation of instant reconfiguration effect or process delays would affect the effectiveness of the management. By setting the management interval to 30 seconds, the authors in [57] observed that under sustained resource demands, a VM needs minutes to get its performance stabilize after memory reconfiguration. Similar delayed effect can also be observed in CPU reconfiguration [56], partially due to the backlog of requests in prior intervals. The difficulty in evaluating the immediate output of resource allocations makes the modeling of application performance even harder.

1.2.4 Issues in Large Scale Cloud Management.

In a cloud, hosted applications such as multi-tier websites and parallel computing programs may run on a group of VMs that span multiple physical hosts. These VMs form a *resource pool*. The resource management of these applications requires that the resource pool should obtain sufficient resources and these resources are properly distributed to each VM. These multi-VM applications usually involve synchronous multi-stage execution in which one stage is blocked until the completion of previous stages. As a result, the performance of the application needs the coordination of all the physical machines that host the virtual cluster. Due to initial placement and load balancing, the actual deployment of these VMs can show an arbitrary topology on physical nodes. As the numbers of physical hosts and VMs increase, the cloud infrastructure is divided into several sub-clusters, each of which is responsible for the resource allocation of one application. These sub-clusters may or may not overlap with each other and the topology can change over time. In such a large scale cloud environment, no centralized management is practical.

1.3 Problem Definition and Objectives

In this dissertation, we aim to design, implement and evaluate a resource management mechanism that delivers stable and adaptive control over cloud resources. In the face of dynamic application demands, the management scheme should leverage cloud elasticity, transparently "adding" or "removing" virtualized resources at fine grain to match the workloads. Resource allocations are in response to the observation of changes in application-level metrics. These metrics include but not limited to performance metrics (e.g. response time and throughput), expenditure metrics (e.g. dollars per hour) and energy consumption metrics (e.g. Joule per hour).

Overall, there are two main requirements in the design of automatic resource management: *transparency* and *assurance*. Transparency requires the management to perform automated resource adjustments during the life time of cloud applications without cloud users' intervention. Transparency implies that the resource management should be able to translate application SLOs to resource requirements. It also requires the management to be adaptive to workload and cloud dynamics. Assurance refers to the guarantee of SLOs in the presence of background management operations. It requires that the management should be responsive to SLO violations and be stable during oscillations.

As stated in Section 1.2, the design of automatic resource management should address the following challenges: (1) It should define a metric that synthesizes multiple application-level metrics and measures a VM's capacity. (2) It should be able to deal with multiple resources. (3) It should employ model-free approaches to handle complex resource to performance relationship. (4) It should be able to provide accurate resource allocation in the presence of process delays. (5) It should scale well assuming no information on actual VM deployment.

1.4 Contributions

To build an automatic cloud resource management system, we have studied the measurement of the capacity of web systems, model-free control of a single virtualized resource, simultaneous control of multiple virtualized resources for system-wide optimization and cluster-wide cloud resource allocation in a large scale. The contribution of this dissertation are as following.

Online web system capacity identification. The first contribution of this dissertation is that we develop a deep understanding in the capacity of online web systems. The understanding of web system capacity is crucial to capacity planning, configuration and QoS-aware resource management. Unlike conventional stress testing approaches measuring server capacity offline using a single performance metric, we propose to use a metric of *productivity index* (PI), which is defined as the ratio of *yield* to *cost*, to measure the system processing capability online. PI is a generic concept that can be applied to different levels to monitor system progress in order to identify if more capacity is needed. We applied the concept of PI to the problem of overload prevention in multi-tier websites. The overload predictor built on the PI metric shows more accurate and responsive overload prevention compared to conventional approaches. The results were published in [59].

Model-free control of virtualized resources. Due to the complex resource to performance relationship in a cloud environment, it is hard to build a mathematical system model that can be used in classical feedback control. Our first attempt to address the issue of the lack of accurate server model is to use a model-free control approach. We extend out previous work on self-tuning fuzzy controller (STFC) for the QoS assurance in physical web servers to the problem of resource allocation in a dynamic cloud environment. By introducing extra layers of output amplification and flexible fuzzy rule selection, the proposed STFC outperform other popular controllers in CPU resource allocation. We also present the design of DynaQoS, an adaptive and multi-purpose QoS provisioning framework based on STFC. This work was published in [58].

Concurrent control of multiple resources for system-wide optimization. In this work, we further improve the automatic resource management in three aspects. First, we extend the automated resource allocation to manage multiple resources. Second, in addition to guaranteeing cloud users' SLA, we also optimize the system-wide performance of the hosting server. Finally, to avoid the performance degradation and oscillation caused by the resource reconfigurations, we optimize the process of reconfiguration operations. We present VCONF, a reinforcement learning-based auto-configuration agent for physical nodes. Reinforcement learning is in nature a model-free and adaptive method and fits well in the system-wide resource allocation. To deal with multiple resources and improve the scalability and adaptability of the learning approach, we propose a model-based reinforcement learning algorithm that makes efficient use of collected system information. The work was presented in [57].

Cluster-wide cloud resource management. In this work, we present a distributed reinforcement learning approach to the cluster-wide cloud resource management. We decompose the cluster-wide resource allocation problem into sub-problems concerning individual VM resource configurations. The cluster-wide allocation is optimized if individual VMs meet their SLA with a high resource utilization. For scalability, we develop an efficient reinforcement learning approach with continuous state space. For adaptability, we use VM low-level runtime statistics to accommodate workload dynamics. Prototyped in a iBalloon system, the distributed learning approach successfully manages 128 VMs on a 16-node closely correlated cluster. More details can be found in [56].

1.5 Dissertation Organization

The rest of this dissertation is organized as follows.

Chapter 2 gives an overview on existing approaches on capacity management. We start with the resource management in physical systems and then discuss the virtualized resource management in cloud systems.

In Chapter 3, we study the measurement of the capacity of online web systems. We first show that offline measurement of server capacity with a single metric lacks accuracy and responsiveness especially in multi-tier websites. We define a productivity index to monitor system progress, based on which a two-layer coordinated predictor is proposed to infer overload problems and identify the resource bottleneck. We evaluate the effectiveness of the approach in terms of prediction accuracy on a multi-tier E-commerce benchmark and show how accurate capacity prediction can help online admission control.

In Chapter 4, we present a model-free approach to control the CPU allocation in a virtual cluster. We first give motivating examples showing that it is difficult to determine a system model in a dynamic cloud environment. After that, we present the design of the self-tuning fuzzy controller and a two-layer QoS provisioning framework that supports multi-objective and service differentiation. We compare the proposed model-free control approach to three other popular controllers and show that STFC has better control performance.

In Chapter 5, we extend the single resource allocation to multiple resources on a physical node. We first show that complex interplay between multiple resources can affect system-wide performance, and then demonstrate how process delay in resource allocation makes fine-grained resource allocation difficult. To address the issues, we propose a model-based reinforcement learning approach, namely VCONF and elaborate its implementation details. Finally, experiments with multiple heterogeneous applications show VCONF's effectiveness in optimizing system-wide with acceptable performance degradation.

In Chapter 6, we present iBalloon, a cluster-wide cloud resource management system based on distributed reinforcement learning. We first review the challenges in cluster-wide resource management and give details on the design of iBalloon. We discuss the state space definition, feedback signal and a highly efficient adaptive learning engine. In the evaluation, we study the sensitivity of learning parameters, effectiveness of the learning engine, scalability and overhead.

Chapter 7 concludes this dissertation with summaries of our approaches and directions for future work.

Chapter 2

Related Work

2.1 Capacity Identification

Server capacity determination is crucial to the problem of resource planning, configuration, and QoS-aware resource management. Early work on server capacity measurement [8] focused on how to generate synthetic workload to stress test the server capacity. Studies in [18] defined a set of benchmarks for stress testing the basic capacities of streaming servers. Unlike their offline measurement approaches, our approach focuses on online measuring the capability of servers for the purpose of request-specific QoS-aware resource management.

Server capacity measurement is necessary for admission control and QoS-aware resource management. An admission controller should know when to turn away excessive requests, and the overload control mechanism should be invoked whenever the server capacity is reached. Most of the past work employed a single rule of thumb to measure server capacity based on application level metrics such as length of the web server request queue [62], incoming traffic density [15, 16], and request response time [84, 34, 25]. In [84], the authors employed a SEDA structure for response time-based admission control. The architecture has no mechanism for capacity measurement. Instead, it used the target response time to a conservative value so as to simplify the design of their admission controller. In [34], half of the most restrictive request response time guarantee was used as the threshold for controlling the incoming request rate. In [25], a measurement-based admission control approach was based on the execution time of requests. However, they assumed a non-preemptive shortest job first scheduling policy in the database server. As a result, the requests would have predictable processing time, independent of the server load condition. It makes it possible to estimate system utilization by monitoring admitted requests. Most of application servers are run in a processor sharing policy. In such servers, the processing time of a request is affected by other requests in concurrent execution. Even with a time-based server capacity estimate, request response time can no longer be used as a reference to calculate server utilization.

There were other QoS-aware resource management work that measured server capacity based on OS level metrics, such as server CPU utilization [23, 17], or hardware performance counter metrics [26]. However, in multi-tier servers, bottleneck resources may shift from tier to tier due to the dynamics of workload and it is difficult to set threshold values for capacity estimation. Our approach uses a combination of these metrics and does not require specifically setting the threshold values for each metric.

Our work in capacity identification is closely related to [19, 90, 20, 24] in that we use similar statistical models to capture underlying server characteristics. Duan and Batu proposed to use synopses in forecasting future event based on historic data [24]. Cohen et al. proposed to use TAN in computer systems [19] and Zhang et al. [90] improved the model accuracy by maintaining an ensembles of models, In [20], Cohen, et al. suggested to use the model to generate system signatures for the purpose of performance problem diagnosis. Our approach is different from theirs in the following aspects. First, we aim at real-time server capacity measurement, while theirs targeted at recursive problem identification. Second, they developed correlation for busy servers rather than overloaded systems. After determining the maximum concurrent level, they set steady state workload at 50-60% of the maximum level. Most importantly, we use multiple synopses for multi-tiers. The prediction results from the synopses are combined together to identify server capacity as well as the bottleneck tier. Wildstrom *et al.* also employed a similar idea using system level metrics for high level decision making [86]. However, their goal was to maximize throughput by reconfiguring hardware under different traffics rather than overload prevention. We used a hardware metrics based derived index to monitor system health instead of simply using OS metrics for workload identification.

2.2 Autonomic Management in Traditional Systems

Early work in autonomic computing [1] aimed to develop computer systems of self-management to overcome the rapidly growing complexity of system management. Recent work often focuses on the design and implementation of self-healing, self-optimization and self-configuration systems.

Self-healing systems automatically discover and correct faults. In [21, 91], Cohen, et al. suggested to use a machine learning model to generate system signatures for the purpose of performance problem diagnosis. They correlated system low-level metrics to high-level performance states. By monitoring sensor readings, the statistical approach was able to narrow down possible faults. In [59], we defined a performance index to measure the system health based on hardware performance counters. A bayesian network model was assumed to automatically map hardware events to system overload state. Studies in [11] reduced downtime of J2EE applications by rapidly and automatically recovering from transient and intermittent software failures, without requiring application modifications.

Self-optimization systems automatically monitor and control resources to ensure optimal performance with respect to defined requirements. Control theory has recently been applied in computer systems for performance optimization. Similar self-tuning adaptive controller were designed in [38, 40] for multi-tier web sites and storage systems. There are other efforts towards automatically allocating resources in a fine grain to individual requests using fuzzy control [80, 44].

Self-configuration systems automatically adapt software parameters, hardware resources for the purpose of correct function or better performance. In [66], *AutoBash* leveraged causal tracking support in Linux to automate tedious parts of fixing a mis-configuration. Chronus in [85] used checkpoint and rollback for configuration management to diagnose kernel bugs.

2.3 Automated Cloud Resource Management

Cloud computing allows cost-efficient server consolidation to increase system utilization and reduce cost. Resource management of virtualized servers is an important and challenging task, especially when dealing with fluctuating workloads and performance interference. Recent work demonstrated the feasibility of statistical analysis, control theory and reinforcement learning to automatic virtual server resource allocation to some extent.

2.3.1 Single Resource Management

Early work [52, 65] focused on the tuning of the CPU resource only. Padala, et al. employed a proportional controller to allocate CPU shares to VM-based multi-tier applications [52]. This approach assumes non-work-conserving CPU mode and no interferece between co-hosted VMs, which can lead to resource under-provisioning. Recent work [37] enhanced traditional control theory with Kalman filters for stability and adaptability. But the work remains under the assumption of CPU allocation. The authors in [65] applied domain knowledge guided regression analysis for CPU allocation in database servers. The method is hardly applicable to other applications in which domain knowledge is not available.

The allocation of memory is more challenging. The work in [31] dynamically controlled the VM's memory allocation based on memory utilization. Their approach is application specific, in which the Apache web server optimizes its memory usage by freeing unused `httpd` processes. For other applications like MySQL database, the program tends to cache data aggressively. The calculation of the memory utilization for VMs hosting these applications is much more difficult. Xen employs Self-Ballooning [46] to do dynamic memory allocation. It estimates the VM's memory requirement based on OS-reported metric: `Commited_AS`. It is effective expanding a VM under memory pressures, but not being able to shrink the memory appropriately. More accurate estimation of the actively used memory (i.e. the working set size) can be obtained by either monitoring the disk I/O [35] or tracking the memory miss curves [92]. However, these event-driven updates of memory information can not promptly shrink the memory size during memory idleness.

2.3.2 Multiple Resources Management

Automatic allocation of multiple resources [51] or for multiple objectives [42] poses challenges in the design of the management scheme. Complicated relationship between resource and performance and often contradicted objectives prevent many work from being automatic but heuristic. Padala, et al. applied an auto-regressive-moving-average (ARMA) model with success to represent the allocation to application performance relationship. They used a MIMO controller to automatically allocate CPU share and I/O bandwidth to multiple VMs. However, the ARMA model may not be effective under steady workload because the recursive least square (RLS) method is effective only when there is enough steepness between two consecutive measurements. The authors also rely on the assumption that drastic variations in workloads that cause significant changes in the model parameters are rare, which limits the applicability of this approach to wider range of platforms. Most importantly, the cost function which directs the resource allocations does not emphersize on the release of unused resources. In another words, the proposed approach can not properly shrink the VM if needed.

2.4 Reinforcement Learning in Autonomic Control

Different from the above approaches in designing a self-managed system, RL offers tremendous potential benefits in autonomic computing. Recently, RL has been successfully applied to automatic application parameter tuning [10, 14], optimal server allocation [68] and self-optimizing memory controller design [33]. Autonomous resource management in cloud systems introduces unique requirements and challenges in RL-based automation, due to dynamic resource demand, changing topology and frequent VM interference. More importantly, user-perceived quality of service should also be guaranteed. The RL-based methods should be scalable and highly adaptive. We attempted to apply RL in host-wide VM resource management [57]. We addressed the scalability and adaptability issues using model-based RL. However, the complexity of training and maintaining the models for the systems under different scenarios becomes prohibitively expensive when the number of VMs increases. In contrast, we fundamentally change the way resource allocation was perceived in iBalloon [56]. In a distributed learning process, iBalloon demonstrated a scalability up to 128 correlated VMs on 16 nodes under work-conserving mode.

Chapter 3

Online Web Systems Capacity Identification

3.1 Introduction

Understanding of server capacity is crucial to server capacity planning, configuration and QoS-aware resource management. It is known that a server can be run in one of the three states: underloaded, saturated, and overloaded. When the server is underloaded, its throughput grows with the increase of input traffic rate until a saturation point is reached. The saturated throughput may not stay unchanged when the input rate continues to increase. It may drop sharply due to resource contention and algorithmic overhead for load management [29]. Knowledge about the server capacity could help measurement-based admission controller in the front-end to regulate the input traffic rate so as to prevent the server from running in an overloaded state. Moreover, for input traffic of multi-class requests, server capacity information can also be used by a back-end scheduler to calculate the portion of the capacity to be allocated to each class for service differentiation and QoS provisioning [25, 81, 88].

An industry standard approach to server capacity measurement is offline stress-testing using benchmarks [18]. It views the server as a blackbox and observes the change of server performance in terms of application-level metrics like response time and throughput with the increase of input load. It approximates server capacity to be the saturated throughput or the system throughput when the observed response time starts to rise abruptly. These offline profiling approaches are limited to systems with static resource configuration. They cannot be applied to today's highly reliable and available servers that are capable of dynamic resource configuration through techniques like hot-swapping and dynamic frequency/voltage scaling [93].

Application-level performance metrics like response time and throughput are good intuitive measures. However, they have limitations in accuracy and timeliness when they are used for *fine-grained* QoS-aware resource management. It is known that requests of an e-commerce transaction have very different processing times and the times also tend to change with server load condition. As a result, request-specific response time becomes an ill-defined performance measure in stress-testing of server capacity. There were studies on the use of mean response time to characterize the server load change in statistics. Welsh and Culler showed that 90th-95th percentile response time represented the shape of response time curve more accurately, in comparison with average or maximum time [84]. However, setting a request-specific response time value for admission control remains non-trivial. In [47], Mogul presented a case that a mis-configuration of the response time threshold could possibly cause the system to enter a live-lock state. In practice, the threshold is often set conservatively. For example, Blanquer *et al.* [34] set a threshold to a half of the target response time of the most restrictive requests for the admission controller to regulate the incoming traffic rate. Such a conservative estimation of the server capacity by setting a low threshold value is equivalent to resource over-provisioning.

Besides the limitation in accuracy, server processing capability measured in application-level response time may not be a timely measure for fine-grained resource management. The observed response time of past requests may mislead the front-end admission controller to wrong decisions because of the presence of long deadtime of requests in a multi-tier website. That is, there is a non-negligible delay from the time a request is admitted to the time its response is observed. Processing tasks of the request could be queueing blocked in many places, particularly when a system is heavily loaded.

In a multi-tier website, processing of a request often involves multiple system components in different tiers. Saturation of the system in the processing of one type of requests may not necessarily mean it cannot handle other requests. Bottleneck may also shift dynamically. Response-time based server capacity measurement provides little insight into constrained resources.

These motivated us to develop an online capacity measurement approach, based on low-level system running statistics. Modern processors are all equipped with a set of performance monitoring registers to record detailed hardware-level system information during the execution time of each application. The information includes a large group of parameters like instruction mix, rate of execution, memory access behaviors and branch prediction accuracy [63]. Together, they define a system internal running state and reflect aggregated effects of the requests in concurrent execution. Questions are how to define a small group of relevant parameters to characterize the system load condition accurately, how to map them onto a high level system overload/underload status, and more importantly how to identify bottleneck resource when the system becomes overloaded.

In this chapter, we present effective and efficient solutions to these questions. Specifically, we develop models involving a small set of hardware performance counter metrics to characterize the system state of each server. We further develop a two-tier coordinated real-time classification approach to infer system overload/underload state and identify resource bottleneck. We evaluated the approach in a two-tier Tomcast/MySQL website using TPC-W benchmark. Experimental results demonstrated its effectiveness and efficiency.

3.2 Limitations of A Single Response Time Metric

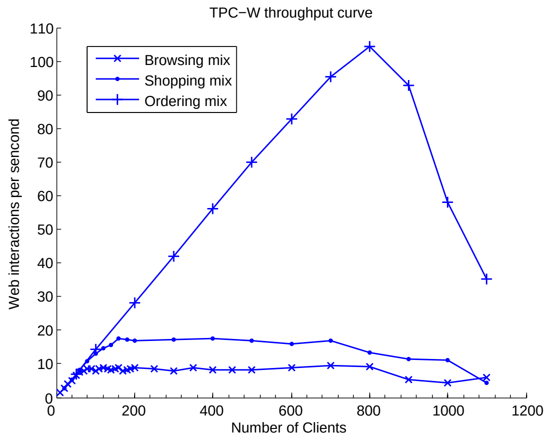
Response time is an application-level intuitive metric for understanding server capacity and user-experienced service quality. However, it is insufficient for the design of request-specific admission control and fine-grained QoS-aware resource management, particularly in multi-tier websites. In this section, we give a brief overview of the dynamics of websites and show limitations of the metric.

3.2.1 Dynamics of a Multi-tier Website

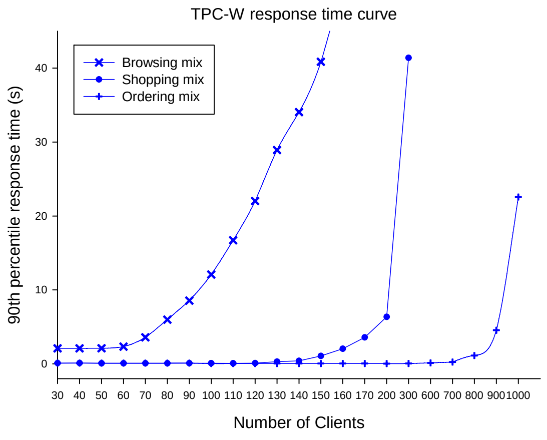
In its simplest form, a website consists of a web server in the front-end, a database server in the back-end, and an application server in the middle to implement the application logic. A configuration example is a Tomcat servlet engine [75] for combined web and application servers and a MySQL [48] for the database server.

Processing of transactional requests often goes through four phases: web protocol parsing, application servlet execution, database connection establishment, and SQL query processing. They are synchronous in the sense that one phase is not finished without the completion of the subsequent phases.

Servers often deploy a multi-threaded processing model to process multiple requests simultaneously. In Tomcat, concurrency is set by a group of system configuration parameters regarding of the maximum number of threads to be run in parallel. Requests will be queueing blocked, if there are no available threads. Requests in



(a) Throughput



(b) Response time

Figure 3.1: Performance of a website in different TPC-W traffic mixes.

processing could also be blocked, waiting for connections to the database server.

In the database server, SQL queries generated by different servlets are not necessarily executed in the same order as they arrive. Because they are executed as a batch of *interleaved* queries, requests in processing may be even been blocked inside the database server.

3.2.2 Website Capacity Identification

In general, a transaction processing system has a saturation point (upper bound) of the throughput the system could produce, as its load increases. After the "upper bound" is reached, the system throughput will either drop because of thrashing or maintain at a saturation level, but with decreased service quality [29]. In order to fully utilize the system resource, admission control must be applied before the system reaches this saturation bound.

We conducted experiments in a typical Tomcat/MySQL website setting, using TPC-W benchmark [72]; Please see Section 5 for details of the test-bed and TPC-W benchmark. TPC-W defines three input traffic mixes: browsing, shopping and ordering, with different request profiles. Table Table 3.1 summarizes the profile of each mix.

Table 3.1: Request composition in TPC-W.

	Browsing	Shopping	Ordering
Browsing request	95%	80%	50%
Ordering request	5%	20%	50%

Figure Figure 3.1(a) shows the throughput curve of these three mixes, in terms of web interactions per second (WIPS), as the number of concurrent clients increases. It is expected that the website would have different processing capacities in different input traffic mixes. With an input of ordering mix, the system throughput drops sharply after it goes beyond the system capacity. In contrast, the throughput stabilizes for the browsing and shopping mixes. This is because browsing related requests tend to put more pressure on the back-end database server, while ordering requests more likely cause CPU overload on the front-end application server. The figure also demonstrates that the bottleneck tend to shift with the change of input patterns.

Figure Figure 3.1(b) shows the 90th percentile response time under different input traffic patterns and different load conditions. The figure shows that for inputs in a browsing mix, the response time increases sharply when the input load goes beyond the system capacity, although the throughput remains unchanged.

Measurement-based admission control needs an online system performance metric to represent the current system load condition. Request response time is a widely used intuitive system load indicator and the metric is easy to monitor online. However, response time-based approach has limitations:

1. It is hard to find "good" thresholds that differentiate "underload" and "overload" system states, because different requests have a large variety of response

times and their execution times varies in different load conditions.

2. Response time of a request can not be measured until the request is completed. It reflects the system status in past time windows, rather than the system's current load condition. Figure Figure 3.2 shows the change of response time in an experiment, in which we injected load spikes at time of 1800th and 5400th second. From the microscopic view of the plot, we can see that the spike at the 5400th second can not be detected in 10 seconds based on response time.

3. Request processing involves many resources in different tiers. The response time metric provides little insight into the bottleneck tier or constrained resources. Since different types of requests put pressure on different tiers of the system, it is possible that, under heavy load, the system's resource bottleneck shifts from one tier to another when the input traffic pattern changes. Using request response time as a system load indicator masks the underlying system load dynamics, and hinders the efficiency of admission control.

Response time measures the system processing capacity based on application-level observation. An alternative is to measure the capacity in lower level system performance metrics.

3.3 Lower Level System Performance Metrics

A system provides a rich set of performance metrics in both hardware and Operating System (OS) levels. Their statistics represent the internal performance states at run-time. Each internal state contributes to a high-level "underload" or "overload" state. Identifying the system state using lower level performance metrics poses three challenges: (1) What metrics should be used to characterize the high level perfor-

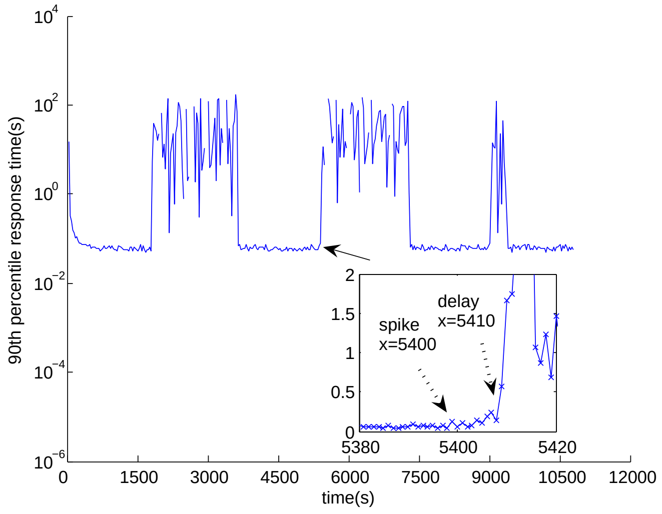


Figure 3.2: Response times in transient spike

mance state; (2) How to infer high level performance states such as "underload" and "overload" from the statistics of the metrics; (3) How to identify the bottleneck tier in a multi-tier website, based on the runtime statistics of each tier. We will discuss the first two challenges in this section and leave the third in Section 4.

3.3.1 Revisit of the Concept of Capacity

System capacity often refers to the maximum amount of work that can be completed during a certain period of time. We refer to the amount of completed work as *yield* and the amount of resource consumed during the time as *cost*. An overloaded system means that its cost keeps increasing but with stagnated or compromised yield. We define a metric of *productivity index* (PI) as the ratio of yield to cost and use it to measure the system processing capability:

$$PI = \frac{Yield}{Cost}.$$

This is a generic concept and *yield* and *cost* can be defined at different system abstractions or under different workload scenarios. By defining *yield* to be the number of requests and *cost* as the wall time, *PI* becomes equivalent to application-level throughput. Today's modern processors are all equipped a number of Hardware Performance Counters (HPC) that provide a rich source of statistical information on application execution. This information includes but not limited to memory bus access pattern, cache reference and pipeline execution information. By defining *yield* as instructions-per-cycle and *cost* the stalled CPU cycles or cache miss rate, the *PI* metric reflects the instruction-level productivity.

The concept of productivity can also be defined at OS level. We argue that OS level metrics like CPU utilization may not be a good metric for system performance. The following example shows that CPU utilization fails to reflect application level performance. Consider the following two code segments on a 2.0GHz Pentium 4 machine with 512 KB L2 cache and 512 MB memory.

```
#define NUM_ITERS 10000
double matrix[65536*8];
int stride=8;

void Sequential(void) {
    for(line = 0; line < 65536*8; line += stride)
        for(offset = 0; offset < stride; offset++)
            for(i = 0; i < NUM_ITERS; i++) {
                temp += matrix[line+offset];
}

void Stride(void) {
  for(i = 0; i < NUM_ITERS; i++)
    for(offset = 0; offset < stride; offset++)
      for(line = 0; line < 65536*8; line += stride)
        temp += matrix[line+offset];
}
```

`Sequential()` accesses consecutive memory locations while `Stride()` visits memory

Table 3.2: Different low-level performance under different workloads.

Workload	IPC	L2 (%)	CPU%	User%	Time (s)
Sequential	0.31	0.03	100%	99.7%	41.4
Stride	0.13	0.92	100%	99.5%	230.5

in strides. Table Table 3.2 shows the execution time for each segment and the statistics reported by OS metrics (CPU% and User%) and hardware counters level metrics (IPC (Instruction Per Cycle) and L2 cache miss rate). The OS metrics shows no performance difference between the two programs. In comparison, hardware-level metrics, IPC and L2 miss rate, reflect application-level performance more accurately. More studies about the selection and effectiveness of hardware-level PI will be given in Section 6. In the following, we use hardware-level PI to measure the system capacity.

For online identification, the single PI metric is not enough to identify system state because any change of PI can be either due to the system capacity or the input load change. For example, a decrease in the incoming workload can lead to a smaller value of PI. But a decrease in PI with sustained or increasing workload can only be due to a system overload. During offline classification, we keep increasing client traffic and label the system state as "underload" until PI begins to drop, from which we label the system state as "overload". During the above process, we take snapshots of hardware counter metrics and develop an online model to correlate them to each high-level system state in a machine learning approach. The model makes it possible for online prediction of system state, for a given set of hardware statistics.

In the following, we give the details of the modeling and learning approach.

3.3.2 Definition of Performance Synopsis

We define a *performance synopsis* data structure to represent the correlation between a set of lower-level performance metrics and their corresponding high-level system states. Formally, let $U = \{A_1, ..., A_n\}$ be a set of attribute variables, in which A_i can be any individual hardware counter performance metric such as number of L2 cache misses. Adding a class variable C into U, we have $U^* = \{A_1, ..., A_n, C\}$. The class variable can be any type of system state. In capacity identification, it is a binary variable, taking value of 1 ("overload") or 0 ("underload"). Each attribute A_i can be instantiated by assigning a measured value a_i during each sampling interval. Instantiating all variables in U^* results in an instance u^*.

For a training set $D = \{u_1^*, ..., u_N^*\}$ with N instances, we build a synopsis to capture the relationship between the group of attributes $A_1, \ldots, A_n$ and class C. We denote it by $SYN(\{A_1, ..., A_n\}, C)$.

Attribute Selection

A system often contains a large number of low-level performance metrics that can be measured online. For example, Linux provides over 100 OS-level metrics; Intel CPU contains hardware performance counters for more than 20 parameters. Including too many attributes in a synopsis could be time complexity prohibitive.

Furthermore, irrelevant attributes in a synopsis may even cause a loss of prediction accuracy. It is desirable to select most relevant attributes in the training set to reduce computing complexity and avoid noises. We apply the concept of *information gain* in information theory to evaluate the relevance between each attribute and the class variable. Information gain is the reduction of entropy about the classification of a test class based on the observation of a particular attribute. For an attribute A_i, its information gain in any class of C can be calculated as follows:

$$\begin{aligned} G(C, A_i) &= H(C) - H(C|A_i) \\ &= -\sum_{c \in C} Prob(c) \log_2 Prob(c) + \\ &\quad \sum_{a \in A_i} \sum_{c \in C} Prob(a, c) \log_2 Prob(c|a), \end{aligned}$$

where $H(C)$ is the entropy of class variable and $H(C|A_i)$ is the conditional entropy of class variable given the attribute variable A_i. Attribute selection is an iterative process, in which the most relevant attribute is added to the attribute set each time only if its addition improves synopsis accuracy. The overall accuracy of a synopsis is evaluated by a *10-fold cross validation* method [41].

Construction of Synopsis and Prediction

A synopsis builder is essentially a machine learning algorithm that generates a synopsis from a training set. In the following, we first present the overview of four representative algorithms that are to be used for synopsis construction. Impact of the algorithms on the prediction accuracy will be discussed in Section 3.6.2.

Linear regression (LR): *Linear regression* is a regression method that models the linear relationship between a dependent variable C, independent variables $A_1, ..., A_n$, and a random term ϵ. To build the LR model is to estimated the coefficients of each A_i and ϵ that best fit in the training set D.

Naive bayes (Naive): Bayesian network is a powerful tool to represent joint probability distributions over a set of random variables. It is often made up of two components: a directed acyclic graph B_s and a set of conditional probability tables

B_p. *Naive bayes* is one of the most effective bayesian classifiers. It makes a strong independence assumption: all attributes A_i are conditionally independent given the value of class C.

Tree augmented naive bayes (TAN): Unlike *Naive bayes*, *TAN* allows the generated B_s to represent correlations between attributes $A_1, ..., A_n$ [27]. The correlations between attributes are captured by imposing a tree structure on the naive Bayesian structure.

Support vector machine (SVM): *SVM* performs classification by constructing an $n-1$ dimensional hyperplane that optimally separates the data into two categories. Different from other classifiers, *SVM* is able to find out the maximum separation between the two classes.

For a synopsis trained from a set D, we consider a set of testing instances p^*, each with a similar structure with the instances in D. For each instance p^*, the same training algorithm of the synopsis is re-applied to generate a prediction C' with respect to the class variable C of the instance. We represent the prediction algorithm as function $Predict()$. That is, $C' = Predict(SYN, p^*)$. If $C' = C$, the prediction is correct, otherwise incorrect.

3.4 Two-Level Coordinated Website Capacity Identification

The preceding section shows the modeling and learning processes to correlate lower-level performance metrics to high level system state in a single server. In a multi-tier website, each server has a PI reference for "underload" and "overload" states. Because the bottleneck may shift between tiers, there are two challenges in the website capacity identification: (1) which PI reference should be used to identify the entire system state offline; (2) which synopsis should be used to predict system state online? We give an overview of the two issues and a solution to the first issue in Section 4.1. The rest of this section is about our coordinated learning approach to the second challenge.

3.4.1 Website Capacity Identification Framework

It is expected that the metrics from a bottleneck tier have the strongest correlation to high-level performance. We select the corresponding PI reference as a measure of the website capacity. We define a correlation index $Corr(PI, r)$, in a way similar to [63], between the PI and high level performance metric r (e.g. throughput) over a time period t:

$$Corr(PI, r) = \frac{Cov(PI, r)}{\sigma_{PI} \cdot \sigma_r} = \frac{\sum_{j=1}^{q}(PI_j - \overline{PI})(r_j - \overline{r})}{q \cdot \sigma_{PI} \cdot \sigma_r}$$

where q is the number of (PI, r) pairs sampled during the time t. The correlation index between PI and r is calculated using their means $\overline{PI}$, $\overline{r}$ and standard deviations σ_{PI}, σ_r in the q samples. The PI with the largest *Corr(PI,r)* value will be selected as the measure of the entire system capacity.

Internet traffic contains different types of requests (e.g. browsing and ordering) and their mix may change with time. Variations of the request composition would affect the performance of a multi-tier website and may even lead to bottleneck shift between tiers. Recall that synopses on each tier are constructed based on specific traffic patterns. Intuitively, a synopsis due to a specific workload is unlikely to be accurate for traffic whose bottleneck lies in another tier. We build synopses on each tier for representative workloads. The workload selection will be discussed in Section

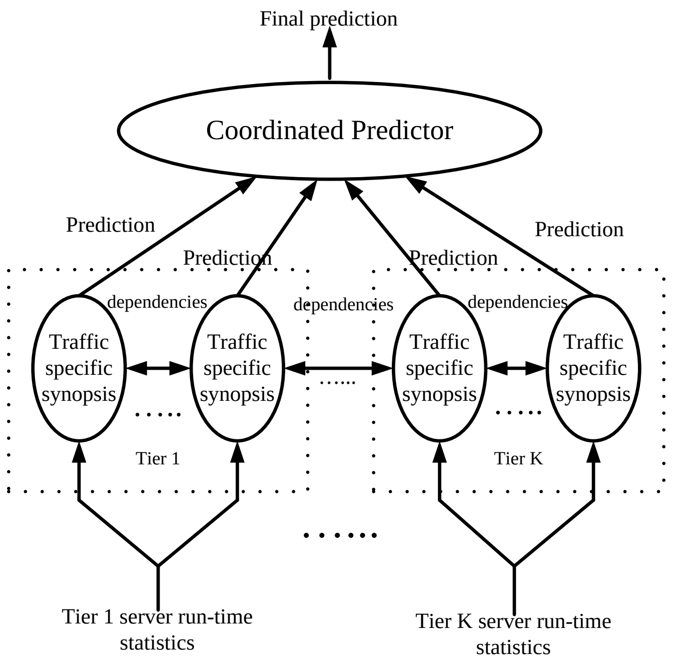

Final prediction
Coordinated Predictor
Prediction
Prediction
Prediction
Prediction
dependencies
dependencies
dependencies
Traffic specific synopsis
Traffic specific synopsis
Traffic specific synopsis
Traffic specific synopsis
Tier 1
Tier K
Tier 1 server run-time statistics
Tier K server run-time statistics

Figure 3.3: The two-level coordinated prediction framework.

5.

For a given set of runtime statistics under a traffic pattern, each workload-specific synopsis will be used to make a prediction. To make a global system state prediction, we propose a two-level coordinated learning scheme which dynamically selects the best synopsis for the given traffic pattern. Following are the details of the scheme.

The capacity measurement framework employs a two-level hierarchical architecture, a group of *performance synopses* in the bottom and a *coordinated predictor* at the top. Figure Figure 3.3 shows the structure. The two-level coordinated prediction architecture takes runtime statistics on each tier machine as inputs. Based on these inputs, individual synopses generate their predictions in regard to system high-level states. Final state prediction will be made in the coordinated predictor by combining these individual predictions.

Although a synopsis is specific to tiers and traffic patterns, the relationship between low level metrics and system state defined by the synopsis remains valid in the presence of workload changes, as long as the bottleneck remains in the same tier. When the workload changes make the bottleneck shifting to another tier, a new synopsis should be selected. The coordinated predictor selects the best synopses dynamically by studying the spatial (synopsis-wise) and temporal patterns among predictions of individual synopses.

Note that a synopsis with less accuracy with regard to certain workloads does not mean that it provides no information for the global system state. Given a workload, predictions from synopses have spatial patterns. For example, synopses might make consistent predictions for certain workloads although the predictions are not correct. Many performance problems manifest not as a single major shift in system behavior but rather as a series of subtle changes. In addition to spatial prediction patterns, temporal patterns among consecutive predictions are also observed in the coordinated predictor.

3.4.2 Coordinated Two-level Predictor

The coordinated predictor is designed as a two-level predictor to capture spatial and temporal patterns in synopses predictions. The coordinated predictor is similar in structure to a branch predictor in superscalar processors [89]. Figure Figure 3.4 shows the structure of the two-level predictor.

The first level is a Global Pattern Table (GPT) which represents synopsis-wise patterns. Each entry in GPT is a Global Pattern Vector (GPV). A GPV is an m-bit vector (m is the number of synopses), and each bit R_i represents the prediction result of corresponding synopsis during a sampling interval τ. That is, $R_i = Predict(SYN_i, p^*_\tau)$. The GPT enumerates all the possible patterns of GPV, thus it has 2^m entries.

The second level are Local History Tables (LHTs) that record the last h prediction results of the specific pattern in GPT. For each of these 2^m patterns, there is a corresponding LHT in the second level which contains the occurrences of different temporal patterns. Each entry of a LHT is referred to as Local History Bits (LHB), denoted by H_c. It is used for making the coordinated prediction. The coordinated prediction is $C'' = \lambda(H_c)$, where λ is the prediction decision function. The length of LHB determines the size of the LHT table. For example, if LHB contains v bits, which records the last v prediction results ($h = v$), the corresponding LHT has 2^v entries.

Along with the two-level predictor for the system state prediction, we also include a simple bottleneck predictor in the coordinated predictor. The bottleneck predictor is implemented by adding an extra Bottleneck Pattern Table (BPT) to the second level. Each entry in the BPT is a Bottleneck Vector (BV) which is indexed by GPV, as well. The bottleneck prediction is $B' = \lambda_b(b_K, ..., b_1)$, where λ_b is the bottleneck decision function.

3.4.3 Training and Prediction

To exploit the spatial and temporal prediction patterns, the coordinated predictor needs to be trained. The training process is to determine the values of LHB H_c in each LHT. Initially, all H_c are set to 0. The values of H_c are learned from all the instances from which each individual synopsis is built. The training process includes the following steps:

1. Given an instance u_i^*, generate predictions from each synopsis. Combining these predictions forms a GPV. Then the GPV, denoted as a binary sequence

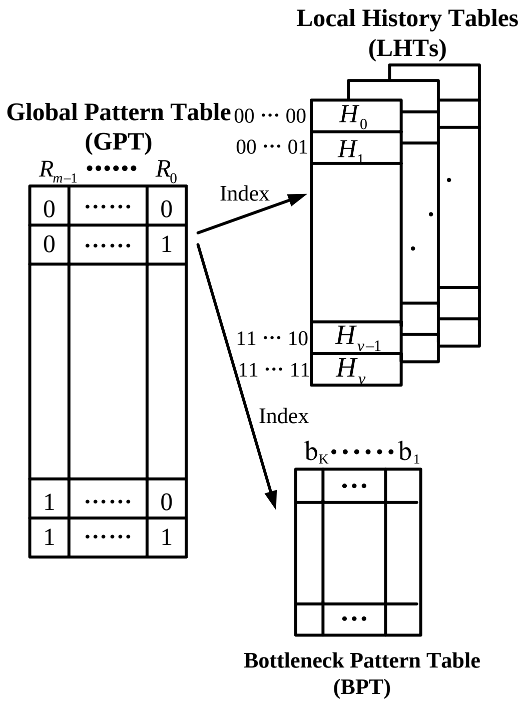


Figure 3.4: The structure of the two-level predictor.

of $< R_{m-1}...R_0 >$, is used to find the corresponding LHT.

2. In the LHT, the local history bits H_c is indexed by last h prediction history. Update the value of the corresponding H_c for each instance u_i^* as follows: If the value of the class variable in u_i^* equals to 1, increase H_c by 1, otherwise decrement by 1.

The training of the bottleneck predictor is similar except that instead of learning H_c values, the values for each $b_K, ..., b_1$ should be trained. For bottleneck identification, we manually augment a training instance u_i^* with information about the bottleneck tier. For example, if the class variable in instance u_i^* has a value of 1 and tier i is the bottleneck for current workload, update b_i as $b_i = b_i + 1$, otherwise $b_i = b_i - 1$.

The coordinated predictor is used to make online global system state predictions as well as bottleneck tier predictions. The bottleneck predictor is only invoked when the system state prediction is 1. For system state prediction, the predictor finds the corresponding H_c according to the current value of GPV. During each sampling interval, the coordinated prediction is made using the prediction decision function $C'' = \lambda(H_c)$, where

$$\lambda(H_c) = \begin{cases} 1 & \text{if } H_c > \delta; \\ \phi(H_c) & \text{if } -\delta \leq H_c \leq \delta; \\ 0 & \text{if } H_c < -\delta, \end{cases}$$

where δ is a threshold for H_c which describes the confidence in H_c making a prediction.

A large δ prevents the predictor from making a prediction unless current spatial and temporal prediction patterns occur a large number of times in previous workloads. Setting δ to a small value relaxes the restriction. For any $\delta > 0$ there exists an interval $[-\delta, \delta]$, in which the predictor is not sure what prediction to make.

We develop two heuristic schemes to select a prediction: an optimistic scheme and a pessimistic scheme. These two schemes are different in function $\phi(H_c)$. The optimistic scheme always makes a prediction of 0 (underload) when $H_c \in [-\delta, \delta]$, while the pessimistic always predicts as 1 (overload).

For the bottleneck predictor, whenever the state predictor predicts as 1, it outputs bottleneck information. The bottleneck decision function is $\lambda_b(b_K, ..., b_1) = \arg\max_i(b_i)$. That is to choose the tier having the largest value in its corresponding bit in $b_K, ..., b_1$ as the bottleneck.

3.4.4 An Example

We use an example to illustrate the two-level predictor. Suppose the website has two tiers: application (AP) and database (DB) tiers, and it takes two different types of input: browsing (B) and ordering (O). There are altogether four ($m = 4$) synopses: AP-O, AP-B, DB-O and DB-B in the GPT, representing synopses for different inputs on different tiers. An coded GPV like (0101) means the predictions of the four synopses are "underload", "overload", "underload", and "overload", respectively. Assume that LHB records the last three overall system state predictions (*i.e.*, h=3), and they are "underload", "overload", "overload", respectively. The corresponding entry H_c in (0101)-indexed LHT is in the address of 110. Suppose the threshold δ is initially set to 5. For any H_c larger than δ, the predictor will forecast an "overload" state.

3.5 Evaluation Methodology

To evaluate the two-level coordinated website capacity measurement, we built a test-bed of multi-tier e-commerce website according to the classic TPC-W benchmark. In our test-bed, the multi-tier website consists of two tiers: front-end application server and back-end database server. Representative workloads conforming TPC-W specifications were thrown to the test-bed. During execution, hardware counter level runtime statistics were collected. For comparison, OS level metrics were also reported.

3.5.1 TPC-W and Workload Selection

TPC-W is a transactional web e-commerce benchmark. Its specification defines 14 different types of requests for an online bookstore service. In our test-bed, we deployed the free Java implementation of TPC-W benchmark from Rice University [61]. TPC-

W defines three traffic mixes: Browsing, Ordering and Shopping, as shown in Table Table 3.1. It classifies web interactions as either Browse or Order depending on whether they involve browsing and searching on the site or whether they play an explicit role in the ordering process.

The primary TPC-W performance metric WIPS is based on the shopping mix, which is the most common workload in an e-commerce website. TPC-W also considers the extreme cases in which the workload is either mostly composed of browsing requests or ordering requests. Experimented with our test-bed, browsing mix is found to put more pressure on database than on application server. For ordering mix, front-end becomes the bottleneck.

We assume that the incoming traffic to a multi-tier website ranges within the above two extreme workloads: Browsing and Ordering. As the characteristic of the workload changes, the bottleneck tier can be either the back-end or the front-end and bottleneck shifting exists. Thus we selected the browsing and ordering mix as the representative workloads for training synopses and the coordinated predictor. The workloads are generated using the Remote Browser Emulator (RBE) shipped with the Rice TPC-W implementation. We modified the RBE to generate the workload needed in training and testing sets. The number of concurrent clients is controlled by the number of Emulated Browsers (EBs).

3.5.2 Training and testing sets

In real scenarios, internet traffic can be either steady or bursty. To generate realistic workloads, we compose the workload generating the training runtime statistics as two parts:

1. **Ramp-up workload**. In ramp-up workloads, we gradually increased concur-

rent client sessions. Because the multi-tier website can serve different numbers of concurrent browsing clients and ordering clients, we increased the workload in different rates. For browsing mix, we started with 20 concurrent clients and incremented 20 clients every 10 minutes up to a limit of 600 concurrent sessions. For ordering mix, we started with 50 clients and added 50 more clients each time until a total of 1500 sessions. For each browsing and ordering mix, we ran the experiments for five hours.

2. **Spike workload**. Spike workload refers to occasional extreme traffic burst. We set the baseline traffic to 80 concurrent shopping clients for both browsing and ordering spikes. Every 30 minutes, we threw a spike workload to the baseline and kept the spike for 10 minutes. Each browsing spike contains 200 browsing clients and each ordering spike has 800 ordering clients. Each experiment also lasted for five hours.

We collected the hardware counter level and OS level runtime statistics on each tier every second. The average statistics over a 30 second interval combined with its corresponding high-level state formed an instance in a training set. The training sets were used to build synopses and tune the coordinated predictor.

Note that although all synopses were trained from the two extreme browsing and ordering mixes, we will show the coordinated predicator works well for traffic of unknown mixes as well. We designed the testing sets as four parts: browsing mix, ordering mix, interleaved mix, and unknown workload mix. The interleaved mix refers to a workload that continues to switch between browsing mix and ordering mix. For the unknown mix, we change the transition probability in RBE to generate workload different from either browsing or ordering mix.

3.5.3 Evaluation Metrics

The key measure of the effectiveness of coordinated predictor is its prediction accuracy in testing sets. Absolute prediction accuracy is the ratio of the number of correctly classified instances over the total number of instances. It depends on the ratio of each class. Instead, we use the Balanced Accuracy (BA) as the metric to evaluate the prediction accuracy. Formally, BA can be defined as:

$$BA = \frac{Prob(C'' = 0|C = 0) + Prob(C'' = 1|C = 1)}{2},$$

where C is the actual value of the class and C'' is the predicted value. Measured by BA, a good predictor should perform well in both classes, independent of the composition of testing sets. To evaluate the prediction accuracy of the two-level predictor, we designed the testing sets mentioned above. We injected approximately 40% to 50% overload instances in each testing set. Thus, any naive method is bounded by a prediction accuracy of 60% at most.

3.5.4 Experiment Settings

We followed the organization of dynamic websites in [5] to build our test-bed except that only one client machine was used to emulate the concurrent clients. The client machine featured a dual AMD 2.10 GHz CPU configuration and 2GB memory. We ensure that the client machine is not the bottleneck by comparing the one client machine experiment with a multiple clients setting. In both settings, the client machine(s) were lightly loaded and the TPC-W performance differences are within 1%. The front-end application server and the back-end database server were configured with Pentium 4 2.0 GHz CPU, 512 MB RAM and Pentium D 2.80 GHz CPU, 1 GB

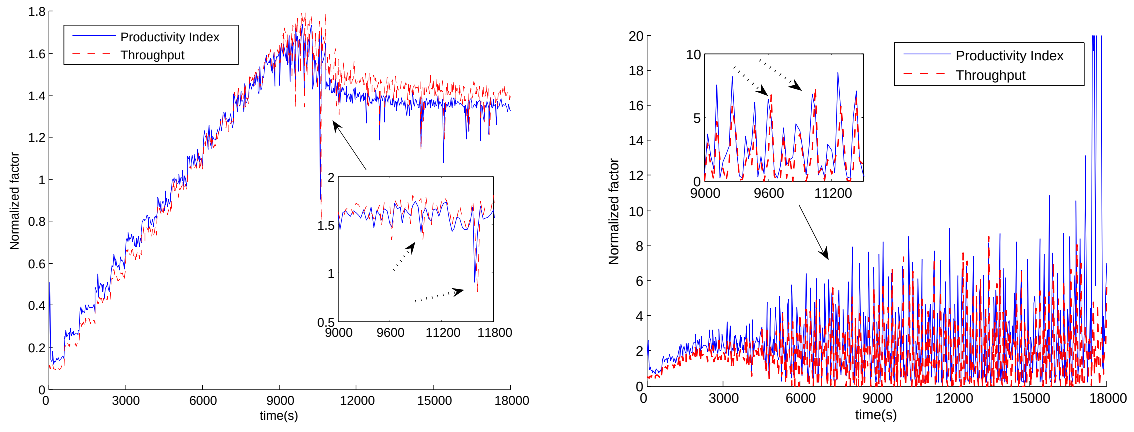
Table 3.3: Collected hardware counter metrics.

Performance counter event	Description
X87_FP_RETIRED	retired uops
X87_FP_UOP	x87 floating point uops
L2_REFERENCE	L2 cache access
L2_MISS	L2 cache miss
CPU_STALL	CPU stalled cycles on any resource
INS_RETIRED	retired instructions
ITLB_REFERENCE	translation lookaside buffer access
ITLB_MISS	translation lookaside buffer miss
RETIRED_MISPRED_BRANCH	retired mispredicted branch

RAM respectively. The CPUs in the servers are based on Intel *NetBurst* architecture and without Hyperthreading technology. All the devices were interconnected by a fast Ethernet network.

The machines ran Fedora Core 6 Linux with kernel 2.6.18. We used Apache Tomcat version 5.5.20 as the application server. For the database server, MySQL standard version 5.0.27 was used. We used `Sysstat` version 7.0.3 to collect 64 OS level metrics. Hardware counter level metrics were recorded by a kernel patch `PerfCtr` [54]. There are software packages, such as OProfile[50], PAPI [53], and PerfSuite[55], which can be used to monitor hardware counter level runtime metrics. Because of their overhead concerns, we wrote a lightweight tool to read hardware counter metrics in all physical CPUs using the global mode in `PerfCtr`. Although the global mode in `Perfctr` only updates performance counter values at regular intervals which may not be accurate enough for small code regions, server programs always run for a long time and management operations are invoked in the granularity of several seconds or minutes. Event counter maintenance in hardware requires no runtime overhead [63] and we limited our tool to minimum functionalities that just initialize and read hardware counters to reduce runtime overheads.

There are 18 performance counter registers in Intel Pentium 4 CPU. Due to



(a) Ordering mix (b) Browsing mix

Figure 3.5: Effectiveness of PI in reflecting high-level performance.

hardware restrictions only 9 registers can be used simultaneously. The performance counter metrics collected in our experiments are listed in Table Table 3.3.

The machine learning algorithms used in our experiments were adapted from WEKA [83] data mining software.

3.6 Experimental Results

3.6.1 Effectiveness of Productivity Index

The first experiment was conducted to show the effectiveness of *productivity index* in reflecting system high-level performance. We took Ordering and Browsing workloads as input and drove the test-bed into an overloaded state. We selected yield and cost metrics according to the correlation measure $Corr$. For an ordering mix input, the front-end server turned out to be the bottleneck and the PI defined as IPC over L2 cache miss rate had the most correlation with the high level performance. For a browsing mix input, database IPC and stalled CPU cycle metrics were selected as yield and cost, respectively.

Table 3.4: Prediction accuracy of individual synopsis tested by Browsing mix.

Specific Synopsis		OS Level				HPC Level			
Workload	Tier	LR	Naive	SVM	TAN	LR	Naive	SVM	TAN
Ordering	APP	0.585	0.500	0.505	0.545	0.570	0.500	0.502	0.505
	DB	0.473	0.500	0.465	0.587	0.439	0.453	0.493	0.646
Browsing	APP	0.635	0.621	0.505	0.603	0.529	0.557	0.540	0.515
	DB	0.604	0.612	0.667	0.635	0.859	0.935	0.957	0.965

Figure Figure 3.5 shows the effectiveness of PI as an indicator of high-level throughput. In order to display PI and throughput curves in a similar scale, we normalized each of their values to their geometric means in different sampling intervals.

Figure Figure 3.5 suggests that for both workloads, the PI and throughput metrics are in high agreement with each other. From the microscopic views, we can see that whenever there is a drop in PI, the corresponding throughput would decrease. Moreover, during some intervals, as pointed out by dotted arrows in the figure, the PI is more responsive than the throughput metric. PI also provides useful information about system-wide performance problems. For example, for the ordering mix input, our test-bed was overloaded due to the application server bottleneck. A drop in PI value suggests decreased IPC and increased L2 cache miss rate. The degraded performance may due to wasted work during context switching when there were too many threads in access to L2 cache in a time multiplexing way.

3.6.2 Individual Prediction Accuracy

The second experiment was designed to demonstrate the prediction accuracy of individual synopsis. A high synopsis accuracy means that the low-level metrics selected are sufficient in representing system internal states and the machine learning algorithm used is capable of correlating low-level metrics to a high-level state.

We tested the prediction accuracy for different level of metrics (e.g. OS level

Table 3.5: Prediction accuracy of individual synopsis tested by Ordering mix.

Specific Synopsis		OS Level				HPC Level			
Workload	Tier	LR	Naive	SVM	TAN	LR	Naive	SVM	TAN
Ordering	APP	0.842	0.928	0.965	0.935	0.805	0.883	0.921	0.952
	DB	0.689	0.932	0.776	0.665	0.746	0.791	0.844	0.840
Browsing	APP	0.583	0.585	0.593	0.547	0.662	0.588	0.588	0.588
	DB	0.545	0.514	0.512	0.572	0.635	0.659	0.662	0.694

and hardware counter level) and using different machine learning algorithms. Table Table 3.4 and Table Table 3.5 summarize the accuracy results. We make several observations from the results:

1. For each testing workload, only the synopsis from the bottleneck tier and built from a similar workload pattern would produce a high prediction accuracy. For example, the synopsis built from a browsing mix on the database server had an accuracy of 0.965 in Table Table 3.4 due to TAN algorithm. But, even with the same learning algorithm, other synopses led to low accuracy. For example, when tested by browsing mix the synopsis built from ordering mix on application server resulted in an accuracy as low as 0.5. By examining the prediction results, we found that this synopsis generated prediction 0 (underload) most of the time.

2. Overall, hardware counter level metrics produced a higher accuracy than OS level metrics. For an ordering mix input, they achieved an accuracy of 0.952 and 0.935, respectively. But for a browsing mix input, the accuracy of OS level metrics dropped down to 0.635. Note that we can not claim hardware level metrics always perform better than OS level metrics. It depends on the workload characteristics. For some workload (ordering mix) both of them are accurate, but hardware level metrics perform significantly better than OS level metrics in some others (browsing mix). The reason is that hardware level metrics provide

Table 3.6: Execution time for each machine learning algorithm.

	LR	Naive	SVM	TAN
Execution time (ms)	90	10	1710	50

more detailed performance information. However, OS level metrics should be considered for I/O intensive workloads because hardware level metrics provide little information on I/O events.

3. Among the machine learning algorithms, SVM and TAN gained highest accuracy in most of the test cases. *Linear regression* performed worst because it can only capture linear correlations. *Naive bayes* performed not as well as TAN. It is because of its strong assumption on the independence of low level parameters.

Table Table 3.6 lists the execution time required to build a synopsis and make a single prediction using different machine learning algorithms. Although SVM has good prediction accuracy, it is computational prohibitive in online performance monitoring. Considering the accuracy and runtime overhead, TAN becomes the best choice for synopsis construction.

In addition to prediction accuracy, *TAN* also provides insights on bottleneck resources. Figure Figure 3.6 shows the TAN structure for the application server synopsis built from ordering workload. Recall that for the ordering mix the front-end CPU is the bottleneck and server overload is due to excessive concurrent requests. From the Bayesian network in Figure Figure 3.6, we can see that hardware counter metrics such as ITLB_REFERENCE, L2_MISS and ITLB_MISS were highly correlated to high-level overload state.

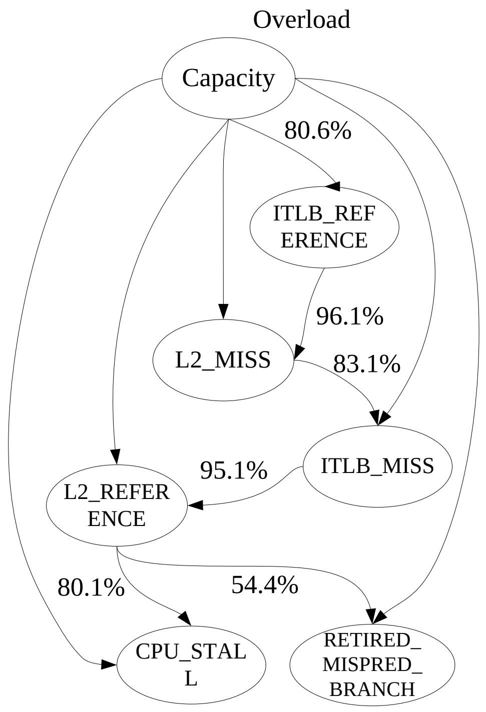


Figure 3.6: Bayesian Network structure for hardware counter metrics.

3.6.3 Coordinated Prediction Accuracy

The third experiment was to demonstrate the overload prediction accuracy and bottleneck identification accuracy of coordinated predictor under different workloads. We used TAN learning algorithm in each synopsis and set the length of history bits to 3. We assumed *optimistic* scheme with a threshold $\delta = 5$.

Figure Figure 3.7 presents the results based on both OS level and hardware counter level metrics. For overload prediction in Figure Figure 3.7(a), similar to individual synopsis accuracy, OS level metrics led to poor accuracy in a browsing mix input. Hardware counter metrics resulted in consistently high prediction accuracy over all the workloads. For a priori known traffic (e.g. ordering mix), the prediction accuracy can be up to 90%. For interleaved workload, which consists of either browsing or ordering mix during any interval, the coordinated predictor still has an accuracy over

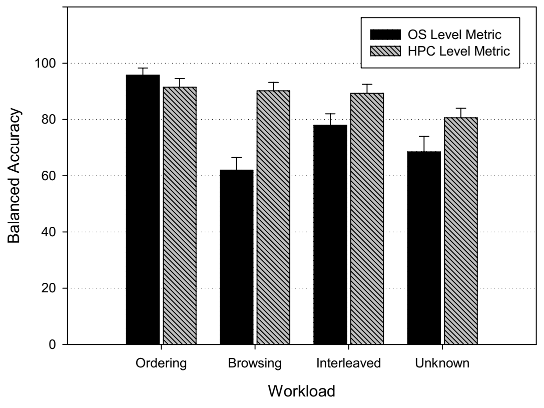
85%. The predictor is robust to workload changes and can maintain high accuracy even in the presence of bottleneck shifting.

It is expected that coordinated predictor would not be able to outperform the best individual synopsis for current workload. Based on spatial and temporal patterns in individual synopses, the predictor actually masks inaccurate synopses and selects the best synopsis for a workload. But for unknown workload, individual synopsis will have a degraded accuracy due to the limitation of supervised learning. Thus, the resulted coordinated accuracy decreased to approximately 80% in unknown workload input, which should be still acceptable.

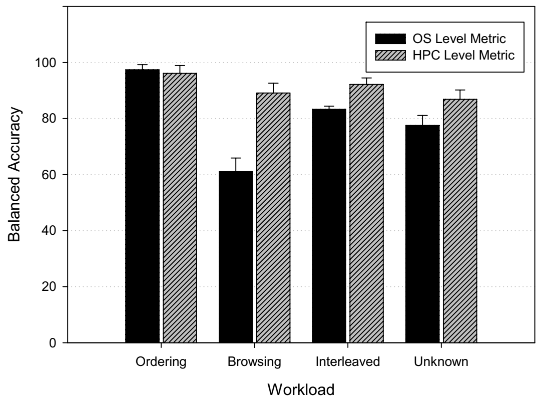
For the bottleneck identification in Figure Figure 3.7(b), the hardware counter level metrics also show consistently good accuracy. It is interesting that the bottleneck prediction accuracy has a similar trend as overload prediction in Figure Figure 3.7(a). This may be due to the similar way the bottleneck identifier exploits the patterns in individual bottleneck prediction.

Recall that the results in Figure Figure 3.7 were obtained under an assumption of optimistic scheme and a 3-bit history. In the following, we evaluated the impact of these two factors. Figure Figure 3.8(a) shows that the schemes had little impact on the coordinated accuracy and there is no single scheme that performs consistently better than the other one. A possible reason is that the spatial and temporal patterns are obvious, the cases that the predictor is not sure are rare.

The length of the history bits controls how many steps the coordinated predictor looks back before making a prediction. Results in Figure Figure 3.8(b) suggest there would be an increased accuracy when history bits be used. In most cases, a single history bit would improve the accuracy by approximately 10%. However, any further history information would lead to only a marginal improvement or even accuracy loss.

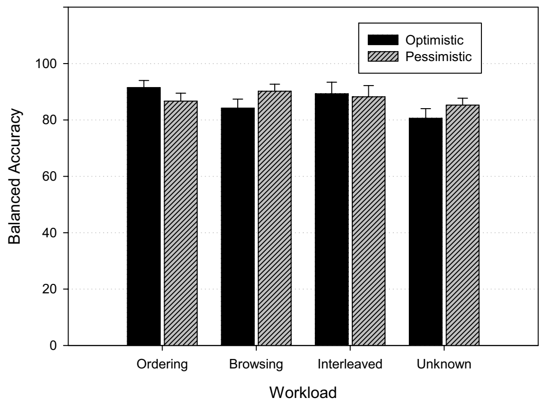


(a) Overload prediction

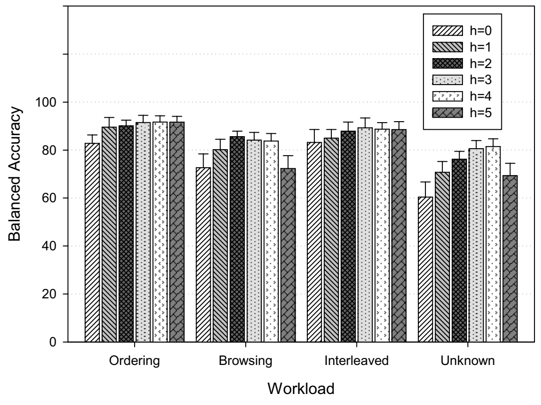


(b) Bottleneck prediction

Figure 3.7: Coordinated prediction accuracy under different workloads.



(a) Impact of schemes



(b) Impact of the length of history bits

Figure 3.8: Impact of design parameters on accuracy.

Table 3.7: Runtime overhead in collection of low-level metrics.

	Throughput loss	Latency increment
OS	2.64%	3.74%
hardware counter	0.49%	0.34%

3.6.4 Runtime Overhead

The last experiment was to investigate the runtime overhead of the predictor. The cost for prediction in different machine learning algorithms was shown in Table Table 3.6. Table Table 3.7 lists the runtime overhead for OS and hardware counter level metrics collection. We normalized the throughput and request latency with respect to the values without metrics collection. The experiments takes an average of 5 executions and each execution lasted 30 minutes. The results show a much lower overhead for the hardware counter metrics collection. The throughput loss and latency increment are within 1% for hardware counter metrics collection.

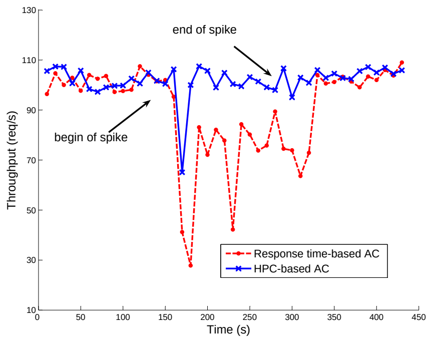
3.7 Application of Capacity Prediction in Online Admission Control

One application of multi-tier website capacity prediction is to guide an admission controller to reject excessive client requests when the incoming traffic exceeds the website capacity. Accurate predictions of the system capacity is crucial to the effectiveness of the admission control. We implemented the two-level capacity predictor in a standalone HTTP proxy residing on a separate machine. The proxy, on which admission controls can be applied, simply relayed the client requests to the front end of the multi-tier website. The proxy collected different levels of performance metrics in a specified interval (a 10-second interval in the remaining experiments), based on which the two-level predictor makes capacity predictions.

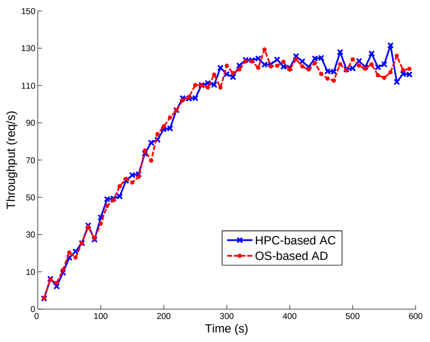
With online admission control, we verified that application level metrics like response time are not reliable for system capacity identification. Figure Figure 3.9(a) plots the throughput of the multi-tier website under a transient spike due to different admission control mechanisms. The baseline traffic was 400 ordering clients. At the 150*th* second, a 600 ordering client spike was generated by another client machine. Figure Figure 3.9(a) compares the hardware performance counter-based admission control with the response time-based one. To isolate the effect of admission control from the traffic variation, we simply instructed the proxy to reject the requests from the IP address generating the spike if an overload is detected. In this way, the accuracy of the capacity prediction became the sole factor that affected the effectiveness of the admission control.

In Figure Figure 3.9(a), we can see that hardware level metrics-based admission control can detect the overload immediately after the arrival of the spike and was able to maintain the throughput at a high level. In contrast, response time-based admission control failed to respond to the overload condition before the spike invaded the system. As a result, the website entered a churn state with up to 70% throughput loss and the overload remained for some time even after the spike's leave.

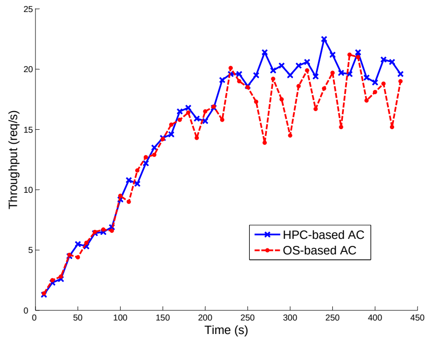
Figure Figure 3.9(b) and Figure Figure 3.9(c) compare the HPC level metrics-based admission control with the OS level metrics-based control under different traffic mixes. Instead of throwing transient spike to the website, we gradually increased the client traffic to overload the website in a step of 50 ordering clients and 10 browsing clients every 30 second. We implemented a simple adaptive rejection rate control based on the following heuristics: increase the rejection rate by 10% if the last system state (in the last 10-second interval) is "overload"; restore to the initial rejection rate if the last state is "underload". The initial rejection rates were set to 15% and 10%



(a) Transient spike



(b) Ordering mix



(c) Browsing mix

Figure 3.9: The effectiveness of admission control using different metrics.

for ordering and browsing mixes respectively.

As discussed in Section 3.6.2, OS level metrics are accurate in determining the system capacity under the ordering mix. In Figure Figure 3.9(b), we see similar admission control effects using HPC and OS level metrics: both effectively rejected excessive requests and stably maintained throughput. In contrast, as shown in Figure Figure 3.9(c), OS level metrics failed to identify system overload as accurately as HPC level metrics under browsing mix, which results in large fluctuations in OS-based admission control.

3.8 Summary

In this chapter, we proposed a two-level coordinated machine learning approach to measuring the multi-tier website capacity based on hardware performance counters. We developed performance synopses to correlate low-level hardware counter metrics with high level system states of each tier. A coordinated predictor was then used to infer system-wide overload/underload state and identify resource bottleneck. Experiments results demonstrate the effectiveness of our approach at less than 0.5% overhead even in the presence of workload changes and bottleneck shifting.

Chapter 4

Model-free Control of A Single Resource

4.1 Introduction

Regulatory control is a promising method for resource allocation, in which a feedback controller enforces service-level objectives while minimizing the resources required. More importantly, if properly designed, this type of control can provide predictable performance with theoretical stability guarantees. In general, a feedback controller applies the *control input* to a target system in order to regulate the *measured output* to the value of a *desired output* [30].

There are many control approaches that have been applied with success to resource allocation in physical servers; see [2, 45, 60, 38, 40] for examples. Recent studies have focused on the application of control approaches for the allocation of virtualized resources in clouds [43, 79, 37, 52, 51]. The cloud adds new challenges to the QoS-oriented resource allocation, in addition to workload dynamics. Different from physical servers, a virtual server may see a varying capacity in the cloud. The dynamics in the capacity can be due to the uncertainties in resource scheduling, opportunistic use of additional market-based resources(e.g. Amazon spot instances [4]) or even the rogue behavior of malicious users [94].

Many existing work used indirect metrics such as workload arrival rate [43, 79] and CPU utilization [37, 52], instead of response time as the measured output. These work relied on the assumption that there are always static relationships between the metrics and response time. The relationships are usually determined either by industry practice or offline testing. Although easier to control, the use of indirect metrics may not be effective in a dynamic cloud environment. In Section 4.2, we show that when the CPU utilization is 80%, the response times of an E-Commerce benchmark can have as large as 150% variations with different capacities. Therefore, with dynamic capacity, resource utilization is not readily translated to application-level performance and models obtained under one capacity setting are likely to be inaccurate for other settings. In practice, response time is a good measure of client-perceived QoS. However, response times behave nonlinearly with respect to resource allocations and are highly dependent on the characteristics of workload, as well as server capacity. This nonlinearity poses challenges to design a stable and accurate controller.

To address the issue of the lack of an accurate server model, the work in [37, 51] applied adaptive control approaches based on model approximation. However, these approaches pose limitations on how fast the workload and the system behavior can change [95]. In [82], we developed a two-layer self-tuning fuzzy control (STFC) approach for QoS assurance in web servers with respect to response time. In this chapter, we extend the STFC approach to resource allocation in virtualized environments by introducing an extra self-tuning output amplification and flexible rule selection mechanism. In comparison with other popular controllers, STFC shows better adaptability and stability. Based on the STFC, we further design a two-layer QoS provisioning framework, DynaQoS, that supports adaptive multi-objective resource allocation and service differentiation.

4.2 Motivating Examples

To build a resource controller realizing a high-level objective, a mathematical model that captures the relationship between the allocated resource and the high-level metric is necessary. Given the model, any deviation of the high-level metric from the desired value can be corrected by applying adjustment in the resource allocation. However, the determination of the system model in a dynamic cloud environment is not trivial. Workload and cloud dynamics can possibly render prior system models invalid and result in poor control performance.

In [6], the authors showed that time-sharing of CPU resources in multiple VMs can provide much more predictable performance than I/O sharing. With advances in multi-core technologies, modern processors are able to embed a number of CPU cores on a single socket. To achieve thread-level parallelism with lower energy cost, heterogeneous CPU architecture and on-chip hardware hyperthreading has gained popularity in modern CPU design. Despite their benefits, they pose significant challenges in VM resource management. “Big” cores are more powerful than “small” cores and hardware threads have distinct performance dependent on whether their sibling threads are executing or not. Current Virtual Machine Monitors such as VMware and Xen, do not consider the underlying architectural differences in VM CPU scheduling. Cloud users may observe different CPU capacities when scheduled with “big” or “small” cores; or with hardware threads from busy or idle cores.

Consider a virtual cluster consisting of 4 VMs executing a mapreduce job to

Table 4.1: Application-level performance difference due to dynamic CPU capacities.

	CPU_PIN	CPU_UNPIN
Bayes classification	668.5s	918.3s

classify approximately 20000 documents into 20 different newsgroups on on a DELL server with 12 CPU cores. Each physical core has two hardware threads which can be scheduled simultaneously. The default CPU scheduling in the Xen hypervisor, referred to *CPU_UNPIN*, allows the two threads from the same core to be scheduled together. In comparison, we experimented with another scheduling scheme, *CPU_PIN*, which ensures that no hardware threads from the same core are scheduled at the same time. It guarantees that each scheduled hardware thread gets the full processing capacity on a core. The experimental results in Table Table 4.1 show that the *CPU_PIN* scheduling reduced the execution time by as large as 37% (reduced from 918.3 second to 668.5 second). This reveals a significant variation of CPU capacity under the same nominal resource allocation.

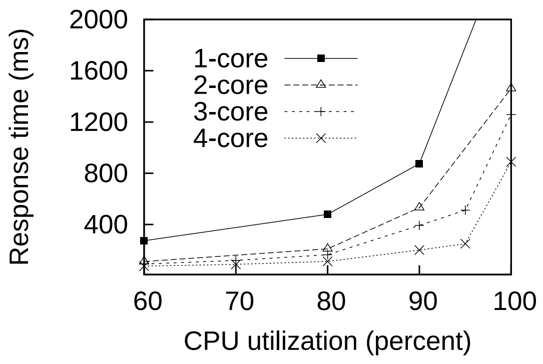
Besides the uncertainties underlying cloud systems, the dynamics in VMs' capacity can also come from market-based accesses to additional compute capacity. Amazon Elastic Compute Cloud (EC2) provides *Spot Instances* [4] as a complementation to *On-demand Instances* and *Reserved Instances*. Different from the other two, Spot Instances make use of unused Amazon EC2 capacity and are charged a much lower spot price. Cloud users bid on spare capacity and run Spot Instances as long as their bids exceed the spot price. Spot price changes with the supply and demand and the instances whose owner's bids are below current spot price will be terminated. If hosted applications are resilient to nondeterministic capacity additions and removals, mixing reserved capacity (i.e. on-demand or reserved instances) with transient capacity (i.e. spot instances) will be a cost-effective way for time-varying workload and limited budget.

However, the nondeterminism in compute capacity poses significant challenges in modeling resource to application performance. Figure Figure 4.1 plots the application performance of TPC-W against the resource utilization (i.e. CPU utilization) under different capacities. We threw 500 shopping clients to the TPC-W virtual cluster and created different levels of capacities by adding or removing VMs from the virtual cluster. For example, a total number of 4 VMs, each with one core, is equavilant to a capacity of 4-core. As shown in Figure Figure 4.1, the relationship between application performance and the CPU utilization changes with capacity. When the CPU utilization is 80%, both response time (Figure Figure 4.1(a)) and throughput (Figure Figure 4.1(b)) show as large as 150% variations with different capacities. With dynamic capacity, resource utilization is not readily translated to application performance. System models obtained under one capacity setting are likely to be inaccurate for other settings. Without an accurate system identification, control-based resource allocation suffers poor performance.

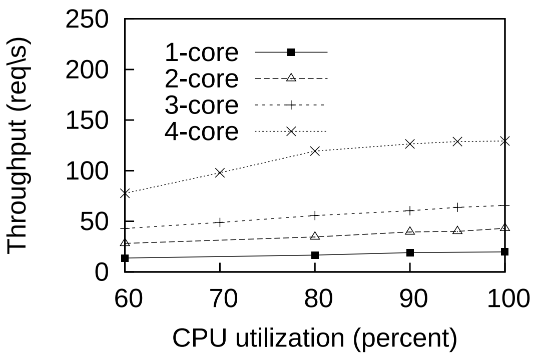
In summary, the discussed challenges motivated us to develop a model-free and self-tuning resource control method that deals with complex resource to performance relationship and dynamic capacity. We propose a novel fuzzy control-based framework, namely DynaQoS, for the management of VM capacity.

4.3 The DynaQoS Framework

In this section, we present the design of DynaQoS, a prototype of the fuzzy control-based VM resource allocation.



(a) Response time



(b) Throughput

Figure 4.1: Different resource-performance relationship due to dynamic capacity.

4.3.1 Design of DynaQoS

As shown in Figure Figure 4.2, DynaQoS is composed of two layers of controllers. The first layer is a group of self-tuning fuzzy controllers (STFC) that control individual objectives. During each control interval, a STFC queries the corresponding QoS profile manager for the reference value of the controlled metric. A QoS monitor periodically reports the achieved value of the metric. The metrics to be controlled can be conventional application-level performance metrics such as response time or throughput; or any user-defined high-level metrics, we show an example of such metrics in Section 4.5. In a cloud environment, more interesting control can be the control of leasing expenses (based on variable resource prices) towards a target of leasing budget, or the control of VM level power consumption below a per VM budget [39]. The STFC takes the difference between the reference value and the achieved one as well as the change of the error as its input and outputs a resource request to the second layer gain scheduler.

When there are multiple control objectives, the second layer gain scheduler aggregates the resource requests from individual STFCs and forms a unified one to be submitted to the cloud resource management API. The aggregation of individual requests is based on the weights (gain) of each STFC in the determination of the final request. The gains are dynamically adjusted according to the control error of STFCs. Service differentiation is necessary if multiple service classes exist and the aggregated resource demand is beyond the available capacity. We define multi-level objectives in the QoS profile manager for each service class. If resource contention is detected and it can not be resolved for a certain number of control intervals, the class with lowest priority modifies its control objective to the next level.

4.3.2 The Self-tuning Fuzzy Controller

Due to workload and cloud dynamics, the relationship between allocated capacity and received service quality exhibits considerable nonlinearities. The relationship can often be linearized at fixed operating points. It is well known that the linear approximation of a nonlinear system is accurate only within the neighborhood of the operating point. Abrupt changes in workload traffics and the nondeterminism in VM capacity can possibly make the simple linearization inappropriate. Instead of modeling the system in mathematical equations, fuzzy control employs the control rules of conditional linguistic statements on the relationship of allocated resource and the high-level objectives [36]. The fuzzy control rules are able to embed human expert's experiences and the rule base is easily updated by adding new knowledge. There are works that applied fuzzy control to QoS guarantees in web server [82] and computer networks [13] with success.

Figure Figure 4.3 illustrates the structure of the Self-tuning Fuzzy Controller. It consists of three components, namely the fuzzy logic controller, the scaling-factor controller and the output amplifier. The resource allocated in control interval $k+1$, denoted by $u(k+1)$, is adjusted according to its error $e(k)$ (i.e., the normalized

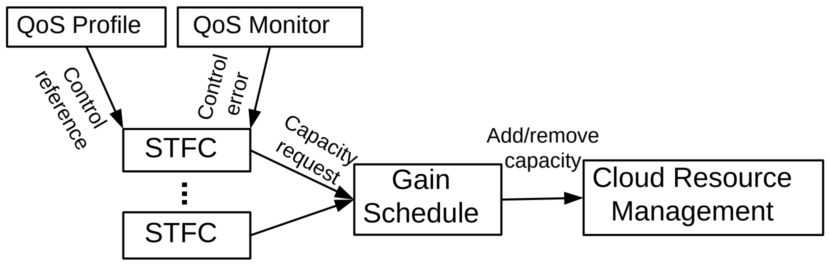


Figure 4.2: The structure of the DynaQoS framework.

difference between the reference value and the achieved one) and change of error $\Delta e(k)$ in previous control interval k using a set of control rules embeded in the fuzzy logic controller. $e(k)$ and $\Delta e(k)$ are calculated using the reference value $r(k)$ and the observed value $y(k)$. For the stability of the control system, we define the normalized error $e(k)$ in a range of $[-1, 1]$:

$$e(k) = \begin{cases} \frac{r(k)-y(k)}{r(k)} & 0 \le y(k) \le 2r(k); \\ -1 & y(k) > 2r(k). \end{cases}$$

Based on these, the controller calculates resource adjustment $\Delta u(k)$ for next control interval. The calculated resource adjustment is then fed into the next layer gain scheduler.

The fuzzy logic controller contains four building blocks. The actual fuzzy logic is implemented as a set of *If-Then* rules about quantified control knowledge about how to adjust the allocation according to $e(k)$ and $\Delta e(k)$. The fuzzification interface converts controller inputs into certainties in numeric values of the input membership functions. The inference mechanism activates the rule-base and applies fuzzy rules according to the fuzzified inputs and generates the fuzzy conclusions for the defuzzification interface. The defuzzification interface converts fuzzy conclusions into the change of allocation in numeric value.

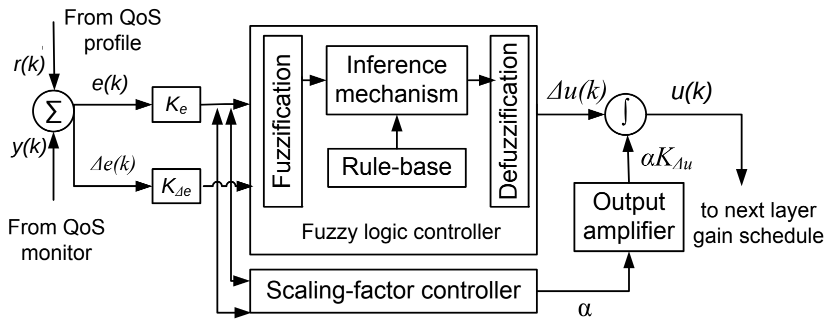


Figure 4.3: The structure of the STFC.

The STFC is built on the static fuzzy logic controller by adding the self-tuning scaling factors and the output amplifier. There are three scaling factors: input factors K_e and $K_{\Delta e}$, output factor α and output amplifier $K_{\Delta u}$. The change of input scaling factors changes the connection of input values to suitable rules, The change of output scaling factor and the amplifier together adjust the amplitude of the control input. The actual inputs of the fuzzy logic controller are $|K_e|e(k)$ and $|K_{\Delta e}|\Delta e(k)$. Thus, the resource allocated to the VM during management interval $k+1$ is

$$u(k+1) = u(k) + \alpha|K_{\Delta u}|\Delta u(k) = \int \alpha K_{\Delta u}\Delta u(k)dk.$$

Design of the rule base

The design objective is to translate human expert's knowledge into a set of control rules to control the resource allocation without a model of the dynamic cloud environment. In the fuzzy logic controller, the control rules are defined using linguistic variables. For brevity, linguistic variables "$e(k)$", "$\Delta e(k)$", and "$\Delta u(k)$" are used to describe $e(k)$, $\Delta e(k)$, and $\Delta u(k)$, respectively. The linguistic variables assume linguistic values NL(negative large), NM (negative medium), NS (negative small), ZE (zero), PS (positive small), PM (positive medium), and PL (positive large).

Figure Figure 4.4(a) gives an simple illustration of typical control effect. In this figure, we identify five zones with different characteristics. Zone 1 and 3 are characterized with opposite signs of $e(k)$ and $\Delta e(k)$, in which the error is self-correcting and the achieved value is moving toward the reference value. Thus, $\Delta u(k)$ needs to be set either to speed up or to slow down current trend. Zone 2 and 4 are characterized with the same signs of $e(k)$ and $\Delta e(k)$, in which the error is not self-correcting and the achieved value is moving away from the reference value. Therefore, $\Delta u(k)$ should be set to reverse current trend. Zone 5 is characterized with rather small magnitudes of $e(k)$ and $\Delta e(k)$. Therefore, the system is at a steady state and $\Delta u(k)$ should be set to maintain current state and correct small deviations from the reference value.The resulted control rules are summarized in Figure Figure 4.4(b). For example, when "$e(k)$" and "$\Delta e(k)$" are NL and PS, "$\Delta u(k)$" is set to PM.

Fuzzification, inference and defuzzification

We take the same design for the membership function and inference mechanism from our previous work; see [82] for details.

At the heart of a fuzzy controller are the membership functions that quantify the *certainty* (between 0 and 1) that an input fall in the corresponding ranges. We select the "triangle" membership function, which is the most widely used in practice. We set the width and height of the "triangle" membership function to be 2/3 and 1 respectively. See our previous work [82] for design details of the membership function. The fuzzification component translates the inputs into corresponding certainty in numeric values of the membership functions. Let $\mu_m(e(k))$ denote the certainty of $e(k)$ of the mth membership function, and $\mu_n(\Delta e(k))$ the certainty of $\Delta e(k)$ of the nth membership function.

The inference mechanism is to determine which rules should be activated and what are conclusions. Let $\mu(m,n)$ denote the certainty of $rule(m,n)$. The *and* operation in the premise is calculated via *minimum*:

$$\mu(m,n) = \min\{\mu_m(e(k)), \mu_n(\Delta e(k))\}.$$

Based on the outputs of the inference mechanism, the defuzzification component calculates the fuzzy controller output, which is a combination of multiple control rules, using "center average" method. Let $b(m,n)$ denote the center of membership function of the consequent of $rule(m,n)$. In this case, it is where the membership function reaches its peak. The fuzzy control output is

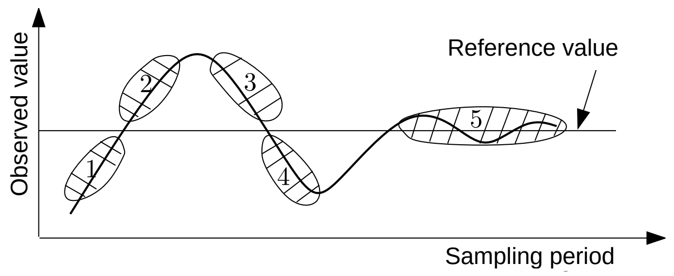
$$\Delta u(k) = \frac{\sum_{m,n} b(m,n) \cdot \mu(m,n)}{\sum_{m,n} \mu(m,n)}.$$

Design of the self-tuning controller

The fuzzy logic controller only defines the basic control rules according to the inputs of $e(k)$ and $\Delta e(k)$. It outputs the sign and magnitude of the allocation adjustment $\Delta u(k)$. With cloud dynamics, there could be a lot of fluctuations in the control effect. To achieve accurate, responsive and stable control, the following practical issues should be addressed:

1. When there are abrupt workload or capacity changes, the control should be responsive enough to correct the resource discrepency within a small number of steps.

2. When there are considerable fluctuations in the control effect, it may be due to two reasons. The fluctuations may come from the inaccuracies of the controller that incurs control overshooting; or it may be due to the process delay [64] of resource allocation. A process delay is the time between the change of resource



(a) The control effect

	"$\Delta u(k)$"		"$\Delta e(k)$"						
			NL	NM	NS	ZE	PS	PM	PL
②	"$e(k)$"	NL	PL	PL	PL	PL	PM	PS	ZE	③
		NM	PL	PL	PL	PM	PS	ZE	NS	
		NS	PL	PL	PM	PS	ZE	NS	NM	
		ZE	PL	PM	PS	ZE	NS	NM	NL	⑤
		PS	PM	PS	ZE	NS	NM	NL	NL	
①		PM	PS	ZE	NS	NM	NL	NL	NL	④
		PL	ZE	NS	NM	NL	NL	NL	NL	

(b) The rule table

Figure 4.4: Design of the fuzzy control rules.

allocation and the actual adjustment effect can be observed in application performance. Both problems can be alleviated by decreasing the control magnitude or prolonging the control interval to stabilize the control effect.

To address the above issues, we design the self-tuning controller to have adaptive output magnitude and flexible control rules. The self-tuning features are realized by dynamically changing the input, output scaling factors and the output amplifier. The output scaling facotr α and the output amplifier $K_{\Delta u(k)}$ together determine the magnitude of the allocation adjustment. In our previous work [82], we used another level of fuzzy controller to adjust the output scaling factor α. However, the output $\Delta u(k)$ of the fuzzy logic controller is within the range of $[-1, 1]$. The change of α has limited effect on the magnitude of the control output. To overcome abrupt workload and capacity changes, the magnitude needs to be changed dynamically based on current conditions. We preserve the adaptive controller of α as in [82] and add a self-tuning output amplifier. The amplifier implements heuristic control knowledge as follows:

$$K_{\Delta u(k)} = |\frac{c}{2} \cdot e(k)|,$$

where c is the current allocation for a specific resource. For example, c can be the cap value of the CPU allocation in a Xen platform. The amplifier follows a heuristic rule that the maximum resource adjustment should not exceed half of current capacity for stability and should be proportional to the control error for adaptability. Note that the direction of the adjustment is still determined by the fuzzy logic.

To address the problem of process delay and control inaccuracies, fuzzy control rules also need to be tuned based on current conditions. Recall that the actual inputs of the fuzzy logic are $K_e e(k)$ and $K_{\Delta e}\Delta e(k)$, K_e and $K_{\Delta e}$ together determine which membership functions or control rules are to be activated. As shown in Figure Figure 4.4(b), small values of K_e and $K_{\Delta e}$ activate rules in the center of the rule table, such as the rules in Zone 5; large values are likely to trigger rules like PL and NL. Observations in the control of real plants suggest that it is often desirable to decrease the control magnitude during fluctuations. Thus, we define K_e and $K_{\Delta e}$ as:

$$K_e(k+1) = (1-\gamma)K_e(k) + \gamma e(k),$$

$$K_{\Delta e}(k+1) = (1-\gamma)K_{\Delta e}(k) - \gamma \Delta e(k),$$

where γ is a discount factor that gives more weight on the observance of recent $e(k)$ and $\Delta e(k)$ while still taking history experiences into consideration. We empirically set γ to 0.8 in the experiments. In Figure Figure 4.4(a), we can see that, during fluctuations the trajectory of control is likely to follow Zone 1 $\rightarrow$ Zone 2 $\rightarrow$ Zone 3 $\rightarrow$ Zone 4. If the pattern is repeated many times, fluctuations exist and $e(k)$ shows as a series of positive and negative values. Gradually, K_e would converge to a small value close to zero, which triggers rules with small or zero magnitude. When the control effect stabilized, if the achieved control deviates from the reference value, K_e will quickly restore to a larger value accumulating $e(k)$ with same signs. The self-tuning scheme works similarly for $\Delta e(k)$ except that $\Delta e(k)$ has the same sign during fluctuations and a subtraction is used to compensate consecutive $\Delta e(k)$. The self-tuning of the control rules also helps mitigate process delays by generating a sequence of small or zero actuations for more stable control.

4.3.3 Scheduling multiple objectives

There exist many control problems in which the consideration of multiple objectives is required, and these objectives may conflict with each other. In cloud computing, a cloud user may want to keep the application level response time and throughput in certain ranges that satisfy SLA objectives. However, the user may be simultaneously required to maintain the power consumption of his or her applications to be below a specified bound. The *Gain schedule* component in the DynaQoS framework implements a weighted scheduling algorithm that synthesizes the outputs from individual STFCs with different objectives. The resulted output is the final resource adjustment request submitted to the cloud resource management. Given individual STFC's outputs $\Delta u_1(k), \ldots, \Delta u_n(k)$ and the corresponding errors $e_1(k), \ldots, e_n(k)$ as inputs, the synthesized adjustment $\Delta u(k)$ is defined as

$$\Delta u(k) = \sum_{i=1}^{n} \Delta u_i(k) \cdot w_i,$$

where $w_i = \frac{|e_i(k)|}{\sum_{j=1}^{n} |e_j(k)|}$.

We assume that there always exists a control solution for the multiple-objective control problem. The gain scheduling algorithm depends on the careful selection of the reference values by the cloud user. If a control solution exists, the algorithm applies dynamic weights to individual STFCs based on their control errors. In the extreme case, the multiple-objective control degrades to a single-objective control, if one objective is satisfied generating near zero control errors.

4.3.4 Realizing service differentiation

Service differentiation is desirable when the aggregated resource demand of multiple service classes is beyond the limit of allocated resources. Although cloud systems allow prompt allocation of resources in response to the increase in client traffic, there are still cases that the total demand can temporally exceeds available capacity. First, the cloud user who owns the cloud application may run out of budget preventing him adding more capacity during a spike load. Second, applications running on the market-based cloud resources may see capacity fluctuations due to the supply and demand of the dynamic capacity. For example, Amazon EC2 users may choose to host applications on a cluster of VMs containing both reserved and spot instances. The spot instances will be terminated if the spot prices exceed the users' bids resulting in a reduction in the total capacity. Finally, complications in cloud resource scheduling and performance interference also contribute to the variation of capacity. For example, results in Section 4.2 show approximately 40% variations in application performance due to scheduling dynamics; the authors in [94] also demonstrated possible CPU cycle stealing between cloud users.

To provide QoS guarantees, we consider the service differentiation to be initiated by individual service classes. When resource contentions are detected, the service class with a lower priority would adapt its SLO (e.g. a response time target) to a lower level. By setting different control objectives, the premium class will receive more resources than the basic class while the basic class avoids starvation maintaining a degraded level of service. We enforce strict priorities between classes. That is the class with a higher priority adapts to a lower level only when the lower priority classes have reached their minimum service levels. To detect resource contentions, DynaQoS follows a simple heuristic rules tracking the statistics of the control performance. If DynaQoS sees a predefined number of serious SLO violations (i.e. $\Delta e(k) < 0$ and $|\Delta e(k)| > \epsilon$) for a certain level of class and the resource adjustment did not correct the control errors (i.e. $\Delta u(k) > 0$), classes with lower priorities would start to adapt to a lower level. Classes at different ranks have the tolerance of different numbers of violations, which ensures that clients with lower priorities will always degrade before the high priority clients. For example, the premium class may only tolerate 10 consecutive violations while the basic class can bear up to 30. When the capacity is limited, the basic class would release the resource first.

4.4 System Implementation

4.4.1 Cloud applications

We selected the TPC-W [73] benchmark as the hosted cloud application for the evaluation of DynaQoS. TPC-W is an E-Commerce benchmark that models after an online book store, which is CPU-intensive and has the database tier as the bottleneck. We employed a three-tier cluster implementation of TPC-W, which consists of an Apache web server (version 1.3.11) and a group of Tomcat (version 5.5.20) application and MySQL (version 5.0.45) database servers. We put the Apache and all the Tomcat servers into one VM forming a unified front-end, and replicated the MySQL server into a number of DB VMs, one MySQL per VM. The DB virtual server farm was further divided into several virtual clusters, each of which was dedicated to a service class. The apache web server accepts and classifies client requests into different classes. It assigns requests from different classes to different DB virtual clusters. We modified the Apache web server to exam the content of the requests and assign different port numbers to different classes. Based on the port number, Apache module `mod_jk` redirects the requests to corresponding tomcat workload balancers which are responsible for individual virtual clusters. The tomcat balancers dispatch the requests within the virtual cluster in a round-robin manner. There may be consistency issues if the requests from a same client session write to different DB VMs. In this chapter, we focus on the evaluation of DynaQoS in resource allocations and leave the issues to future work. To avoid consistency problems, we used read primary browsing mix in TPC-W as the client workload.

We empirically determined that the DB tier was the bottleneck tier under the browsing workload and focused on the CPU allocation to the DB clusters. There are two ways to change the allocation to a DB virtual cluster. One is to change the number of DB VMs in a cluster and the tomcat balancer handles the join and leave of cluster members. Another approach is to have a fixed number of DB VMs and change the CPU allocations to individual VMs. To evaluate DynaQoS in fine-grained resource allocation, we took the second approach.

4.4.2 Testbed

Our testbed consists of a virtual server, client and NFS servers. The physical machines for virtual hosting were two DELL servers with two Intel Xeon X5650 CPUs and 32 GB memory. Each CPU has 6 cores with hyperthreading enabled resulting a total capacity of 24 logical CPUs. The front-end and back-end DB VMs were hosted on separate machines. We configured the front-end VM with 8 core and 4 GB memory. The DB VMs, each with 4 core and 2 GB memory, resided on the other machine. We used a number of client machines each with 8 cores and 8 GB memory to generate workload for the TPC-W. The NFS server used a RAID5 partition to serve the VM disk images. We used Xen version 4.0 as our virtualization environment. `dom0` and guest VMs were running Linux kernel 2.6.32 and 2.6.18, respectively. All the severs were connected by Gigabit Ethernet network.

4.4.3 Implementation of DynaQoS

QoS monitor. We consider the client-perceived response time as the measure of application-level performance. We modified the TPC-W's workload generator to maintain a log of finished requests. A small utility program parses the log to calculate the average response time for every control interval.

QoS profile manager. Each service class works with a QoS profile manager to determine the control objective. The control objectives are specified in terms of a set of desired response times with different levels. For service differentiation, the profile manager also sets the number of SLO violations that can be tolerated by a class before a target adaptation is initiated. For the service differentiation experiment in Section 4.5.3, we considered two classes: *Premium* and *Basic*. They both have three levels of SLO, {1s, 5s, 10s}, and with adaptation thresholds: 10 and 30 violations, respectively.

Self-tuning fuzzy controller. STFC has been implemented as a set user-level daemons in the virtual host (i.e. `dom0` in a Xen environment). It takes the measured application-level performance (from QoS monitor) and the performance objective

(QoS profile manager) as input and outputs the resource adjustment to Xen's management interface. If multiple control objectives exist, two or more STFCs form a unified request. The control interval is set to 30 seconds for all the experiments.

CPU resource allocation. CPU resources are allocated to each DB VM via Xen Credit Scheduler in terms of cap values. A *cap* value represents the upper limit of CPU time can be consumed by a VM in percentage. For a virtual cluster with 4 VMs and each with 4 cores, the CPU allocation can be in the range of $[1, 1600]$. The CPU time is allocated to individual virtual clusters. We assume good load balancing by the Tomcat balancer, thus distribute CPU cap values uniformly within the cluster. All VMs are given the same weight during allocation.

4.5 Experimental Results

4.5.1 Comparing STFC to other popular control methods

Experiments are designed to study the efficacy of DynaQoS in the determination of proper CPU allocations under both static and dynamic workloads. We have also implemented three popular controllers within the DynaQoS framework:

Kalman filer [37] is a data processing method that uses a series of measurement with noises to produce values closer to the true values of the measurement. It is used in [37] to track the utilization of CPU and allocate CPU resources correspondingly to maintain the utilization to a specific value.

Adaptive proportional integral (PI) [52] directly tracks the error of the measured response time and the target and adjusts the CPU allocation to minimize the error. The gains of the proportional and integral parts are set to $|\frac{c}{2} \cdot e(k)|$, similarly as the STFC, to allow adaptive control.

Auto-regressive-moving-average (ARMA) [51] builds a local linear relationship between the allocated CPU resource and the response time with recently collected samples. If response time deviates from the target value, the controller computes the allocation that corrects the error based on the obtained model. The controller is configured to use a second-order ARMA model with a window size of 20.

To measure the performance of DynaQoS, we define a metric, relative deviation $R(e)$, based on root-mean square error:

$$R(e) = \frac{\sqrt{\sum_{k=1}^{n} e(k)^2/n}}{r(k)}.$$

The smaller the $R(e)$, the more the achieved response time concentrates near the target value and better the controller's performance. To compare the performance of different controllers, we take the performance of STFC as a baseline and define the performance difference between STFC and other controller as:

$$PerfDiff = \frac{R(e)_{other} - R(e)_{STFC}}{R(e)_{STFC}}.$$

Response times behave nonlinearly with respect to resource allocations especially when the system is in a busy state. We selected the set point of all the controllers to be 1 second except that we followed the controller in [37] and set the Kalman filter's set point to be 90% CPU utilization, which translates to approximately the 1-second response time under the capacity of 16 cores. In this experiment, we only considered one service class with one virtual cluster. The virtual cluster had 4 DB VMs each with 4 VCPUs and its initial capacity was set to 6 cores (a cap of 600).

Figure Figure 4.5(a) plots the response times of different control methods with static TPC-W workload. The workload was set to 200 browsing clients, each with a mean think time of 1 second. From Figure Figure 4.5(a), we observe that, all the control methods except ARMA can bring the response time close to the 1-second target, but with different deviations. ARMA requires a local model to predict the proper CPU allocation, thus whenever a deviation from the target is detected it needs several control intervals to build a new model. Figure 4.5.1 draws the performance difference of other controllers relative to STFC. STFC outperformed all other controllers by at least 16% with adaptive-PI as the closest competitor.

We are also interested in the adaptability of the controllers under dynamic workload. We instrumented the workload generators of TPC-W to change client traffic levels at run-time. The workload generator reads dynamic traffic levels from a trace file, which models after the real Internet traffic pattern [69]. Figure Figure 4.5(b) plots the response times in a 90-minute period in which the number of clients was changed every 30 intervals. We started with 100 clients and set the client numbers at the $60th$, $90th$, $120th$ and $150th$ interval to be 200, 300, 200 and 100, respectively. From Figure Figure 4.5(b), we observe that, ARMA performed worst among the controllers with a large number of SLO violations. Kalman filter was not responsive to the workload change and failed to bring the response time back to the 1-second target before the workload changed again. Both of STFC and adaptive-PI successfully maintained the response times around the target. Figure Figure 4.5(b) also suggests that STFC is more responsive to the workload change with an average settling time of 3 intervals. In contrast, adaptive-PI had an average settling time of 6 intervals. Figure 4.5.1 reveals that STFC outperformed adaptive-PI by 37% in terms of relative deviation. It is expected that Kalman filter and ARMA incurred large deviations.

To better understand the performance of the controllers under dynamic workload, we also plot the actual CPU allocations (i.e. cap values) in Figure Figure 4.5(c). It shows that Kalman filter is not responsive to the workload change and ARMA is too sensitive to the dynamics. We believe that these two methods can be tuned to fit the system better. However, controllers based on local model approximation impose limitation on how fast workloads can change. Both STFC and adaptive-PI do not assume any models of the underlying system, and were able to adjust the CPU allocations properly. In Figure Figure 4.5(c), we find that, STFC maintains more stable CPU allocations during the period between the workload changes (e.g. between $60th$ and $90th$ intervals). This explains the more stable control performance of STFC in Figure 4.5.1 and is due to the flexible control rule selection in STFC.

4.5.2 Scheduling multiple objectives

In the previous experiment, we set the control objective precisely at 1-second response time. In many problems, relaxing the “point” control objectives to some suboptimal “regions” is also acceptable. This observation makes the simultaneous control of multiple objectives feasible and of practical importance. DynaQoS applies gain scheduling to balance the trade-off between conflicting objectives. In this experiment, we study the simultaneous control of conflicting objectives, application performance and power within the DynaQoS framework.

We assume that individual cloud users are allocated a power budget to limit the power consumption of their applications. There are existing work performed VM-level power measurement with success [39] and we believe that VM-level power budgeting is desirable in future data centers. In this experiment, we tested with only one cloud user and consider the system-wide power as the VM’s consumption. The system-wide power consumption is measured with a WattsUp Pro power meter. The meter records the power consumption every second and we calculate the average power value for each control interval (i.e. 30 second). The more the CPU resources the smaller the response times but the larger the power consumption. The set points were set to 1 second and 250 watt for the response time and power budget, respectively.

Figure Figure 4.7 plots the response time and power consumption during the control. Before the $30th$ interval, the cloud application contributed most to the power consumption and there existed a balance point that generating acceptable performance for both objectives. DynaQoS successfully identified the balance point and stabilize the response time and power consumption at approximately $800ms$ and $190w$, respectively. Starting the $30th$ interval, we launched background jobs in the host consuming considerable power. In this way, we emulate the circumstances in which some other jobs belonging to the same cloud user eat a lot of power and the user needs to limit the power usage by the cloud application. From the figure, we can see that the combined power consumption immediately exceeded and budget and DynaQoS was able to contain the consumption within the budget by reducing the CPU allocation to the cloud application. When the response time or the power deviated from the target, DynaQoS give more weight to the corresponding STFC. With current settings of the objectives, DynaQoS was able to brought both response time and power close to their targets with stable performance. Once the background jobs ended, DynaQoS returned back to the ($800ms$, $190w$) balance point.

4.5.3 Service differentiation

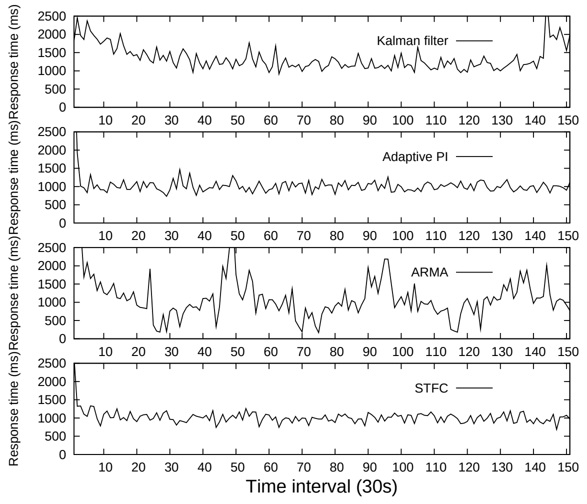
In this section, we investigate the effectiveness of DynaQoS in providing differentiated services to two client classes, *Premium* and *Basic*. DynaQoS applies target adaptation if resource contention is detected. We compare the target adaptation of DynaQoS to a strict differentiation policy (STRICT), that is to guarantee the CPU allocation of the premium class and provide best-effort service to the basic class. To prevent resource starvation of the basic class, we reserve 2-core's capacity to the basic class's virtual cluster. We configured the two classes each with a cluster of 4 DB VMs. Each cluster had 16 VCPUs and can use up to 16 physical CPU resources in theory. The client concurrency levels were set to 200 browsing clients for both classes and resulted in an aggregate CPU demand of approximately 20 CPUs. The server hosting the virtual clusters has a capability of 24 CPUs, thus no service differentiation is needed if the virtual clusters can use the CPU resource freely. We emulated the change in the CPU capacity by restricting the 32 VCPUs of the clusters to the first 12 physical CPUs at the $20th$ interval. This effectively reduced CPU capacity to 12 CPUs. As discussed in Section 4.2, the capacity change due to scheduling dynamics is possible in current cloud platforms. More importantly, the change in the actual CPU capacity may not be reflected in the nominal CPU allocations.

In Figure 4.5.3, we compare DynaQoS's target adaptation with the STRICT policy. We implemented two variations of the STRICT policy, one with the knowledge of the exact value of the new capacity (*STRICT w/ info*), one without (*STRICT w/o info*). As shown in Figure 4.5.3, DynaQoS was able to detect the resource contention at the $32th$ interval because the premium class had seen 10 serious violations in the response time. It triggered the basic class's target adaptation to the next level, 5 second. The performance of both classes stabilized at the $50th$ interval. The premium class succeeded to maintain the 1-second target and the basic class achieved a response time close to its new target. After we increased the capacity at the $60th$ interval, DynaQoS took 10 intervals to detect the change and reset the target of the basic class back to 1 second. In contrast, *STRICT w/o info* policy failed to detect the capacity change and did not enforce service differentiation.

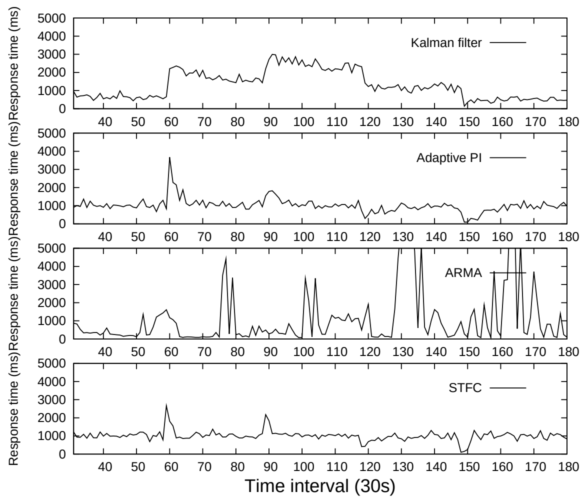
In Figure 4.5.3, we also observe that, with the new capacity information (i.e. 12 CPU), *STRICT w/ info* was able to guarantee the performance of the premium class. But the basic class suffered a 10-second response time compared to 5-second in DynaQoS. An examination of the actual CPU allocation during the contention period revealed that the basic class achieved a 5-second level performance because it obtained more CPU resources than in the *STRICT w/ info* policy. During the contention, DynaQoS kept increase the allocation of both classes until the targets were met. The aggregated CPU allocation in terms of cap values can be beyond the actual capacity (12 CPU). It is equivalent to a work-conserving mode but with bounded allocation to the basic class for the purpose of differentiation. Different from DynaQoS, *STRICT w/ info* enforces that the total allocation is not beyond 12 CPU and the basic class only got an allocation of 2 CPU or whatever was left by the premium class. The non-work-conserving mode in *STRICT w/ info* policy wasted some CPU time which can otherwise be used by the basic class.

4.6 Summary

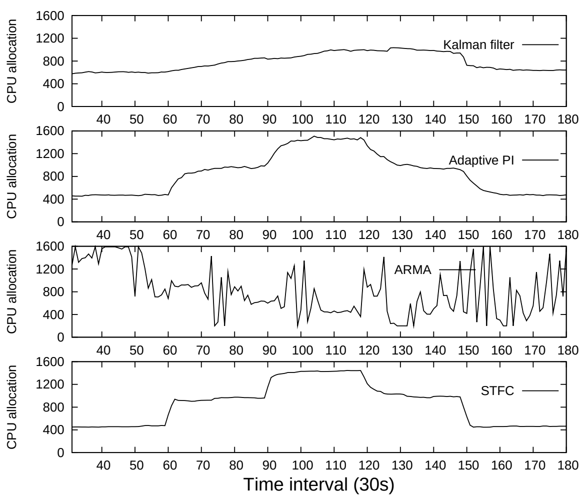
In this chapter, we have proposed a response time-based fuzzy control approach for the allocation of virtualized resources. We develop a self-tuning fuzzy controller with adaptive output amplification and flexible rule selection. Based on the fuzzy controller, we further design a two-layer QoS provisioning framework, DynaQoS, that supports adaptive multi-objective resource allocation and service differentiation. Experiments on a Xen-based cloud testbed and an E-Commerce benchmark show that the fuzzy controller outperformed three popular controllers for CPU resource allocation. DynaQoS also demonstrated its effectiveness in the simultaneous control of performance and power and service differentiation.



(a) Static workload



(b) Dynamic workload



(c) Resource allocation

Figure 4.5: Performance comparison of STFC, Kalman filer, Adaptive-PI and ARMA.

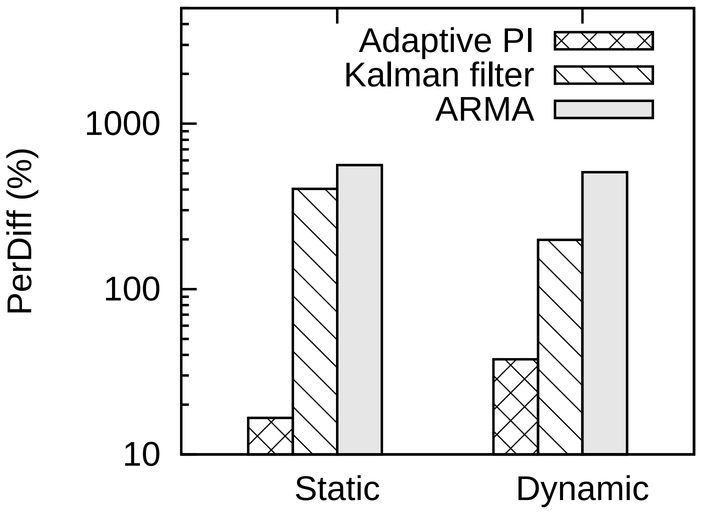


Figure 4.6: Performance comparison of STFC and other controllers in relative deviation.

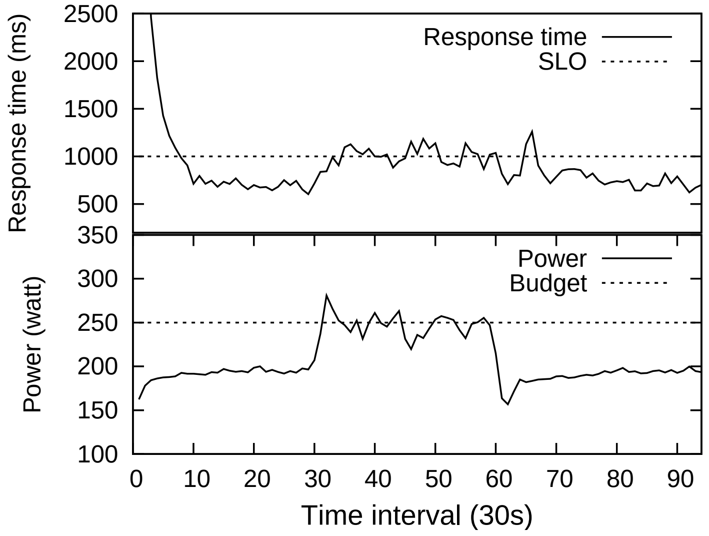
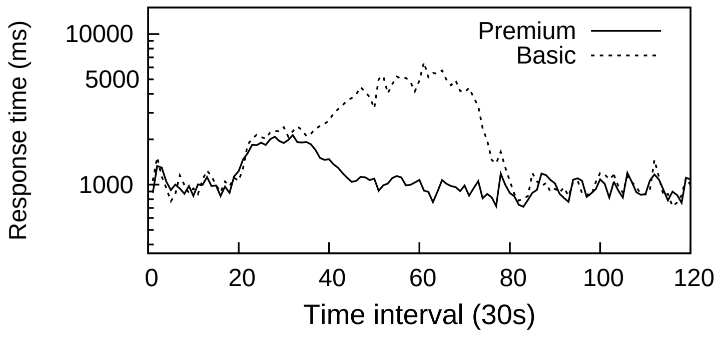
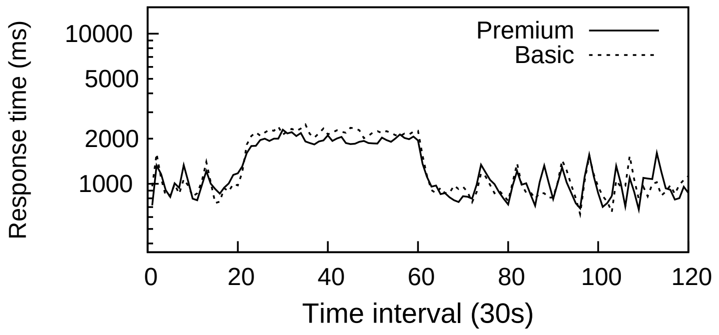


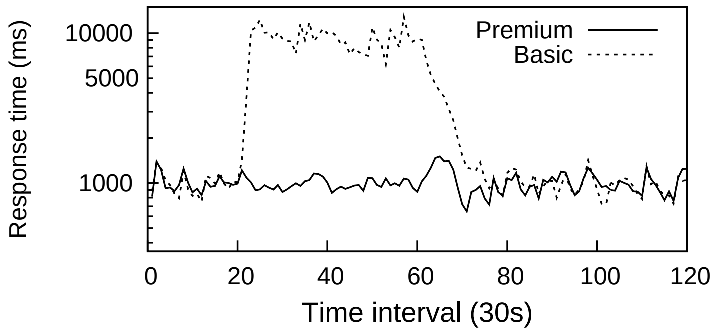
Figure 4.7: Simultaneous control of performance and power.



(a) Target adaptation



(b) STRICT without capacity information



(c) STRICT with capacity information

Figure 4.8: Service differentiation with different methods.

Chapter 5

Concurrent Control of Multiple Resources

5.1 Introduction

Modern virtualization platforms provide a rich set of configurable resource parameters for fine-grained runtime control. Due to time-varying resource demand of typical server applications, it is usually necessary to re-allocate resources to hosted VMs for overall performance. In server consolidation with heterogeneous applications, it is not easy to figure out the besting settings for VMs with distinct resource requirements.

Server virtualization has a key request for performance isolation. In practice, applications sharing the physical server still have chance to interfere with each other. In [49, 28], the authors showed that bad behaviors of one application in a VM could adversely affect the others' in Xen [9] VMs. The problem is not specific to Xen; it can also be observed on other virtualization platforms due to the presence of centralized virtual machine scheduling. Therefore, to optimize system-wide performance, it is desirable to have balanced resource configuration in co-resident VMs preventing rogue

VMs that affect others. Furthermore, there is usually a time gap between resource allocations and their effects on application performance can be observed. We call it a *process delay* or *delayed effect*.

An automated resource management should optimize system-wide performance with balanced VM configurations, even in the systems those are highly dynamic and hard to model. It should also be able to deal with delayed effects. Reinforcement learning (RL) is a process of learning by interactions with dynamic environment, which generates the optimal control policy for a given set of states. It requires no domain knowledge of the controlled system and is able to offset delayed effects by optimizing a long-term goal. RL approaches to the design of computer systems involve several important issues. The application of RL methods is non-trivial due to the exponentially increased state space when systems scale up. In online system management, interaction-based RL policy generation suffers from slow adaptation to new policies.

In this chapter, we propose a RL-based virtual machine auto-configuration agent, namely VCONF. The central design of VCONF is the use of model-based RL algorithms for scalability and adaptability. We define the reward signal based on summarized performance of each VM. By maximizing the long run reward, VCONF automatically directs each virtual machine configuration to a good (if not optimal) one.

5.2 Motivating Examples

In this section, we first briefly review the Xen virtual machine monitor (VMM) and then give some motivating examples showing that why balanced VM configurations are desirable and how delayed effets affect resource allocations.

5.2.1 The Xen Virtual Machine Monitor

A VMM is the lowest level software abstraction running on the actual hardware. It provides isolation between guest OSes and manages access to hardware resources. Xen [9] is a high performance resource-managed VMM. It consists of two components: a hypervisor and a driver domain.

The hypervisor provides the guest OS, also called a guest domain in Xen, the illusion of occupying the actual hardware devices. The hypervisor performs functions such as CPU scheduling, memory mapping and I/O handling for guest domains. The driver domain (dom0) is a privileged VM which manages other guest VMs and executes resource allocation policies. Dom0 hosts unmodified device drivers for VMs; it also has direct access to actual hardware. Xen provides a control interface in the driver domain to manage the available resources to each VM. The following configurable parameters have salient impacts on VM performance.

1. **CPU time.** The Xen VMM uses a credit scheduler to schedule CPU on domains. Each VM is assigned a credit number which statistically determines the portion of processor time allocated to each VM.

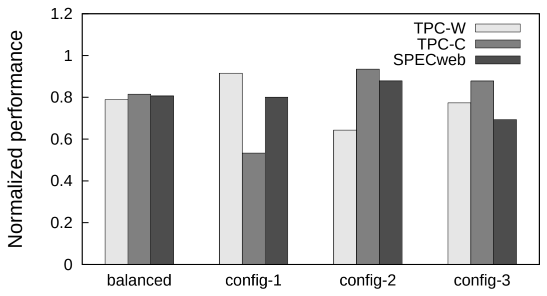
2. **Virtual CPUs.** This parameter determines how many physical CPUs can be used by a VM. The number of virtual CPUs together with the scheduler credit determine the total CPU resource allocated to a VM.

3. **Physical memory.** This parameter controls the amount of memory can be used by a VM. If not set appropriately, the application within the VM may need to communicate with disk frequently, which degrades user-level performance considerably.

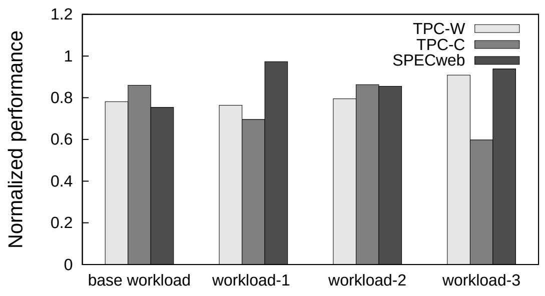
5.2.2 Balanced Configurations

In Xen's implementation, privileged instructions and memory writes are trapped and validated by the hypervisor; I/O interrupts are handled by the VMM and data is transfered to VMs in cooperation with dom0. The involvement of the centralized virtualization layer in guest program execution can also be found in other platforms, such as VMware [76] and Hyper-V [32]. Thus, any bad behavior of one VM may adversely affect the performance of other VMs by depriving the hypervisor and driver domain resources. In [28], the authors showed that for I/O intensive applications, by setting a fixed CPU share, the credit scheduler does not account for the work done for individual VMs in the driver domain. Taking memory and virtual CPU into consideration, the involvement of dom0 and hypervisor in VM execution aggravates the uncertainties in resource to performance mapping. For example, allocating more resource to one VM may result in performance degradation due to the other VMs' impediment caused by resource deallocation.

For example, on a host machine with two quad-core Intel Xeon CPUs and 8 GB memory, we created three VMs running representative server applications: E-Commerce (TPC-W [73]), online transaction processing (TPC-C [74]) and web server benchmark (SPECweb [71]). Details of the testbed and benchmark settings can be found in Section 5.5. Figure Figure 5.1(a) shows the impact of resource configuration on application performance. The throughput for each application is normalized to a reference value resulted from running the application on the host exclusively. The balanced configuration in the form of (time, vcpu, mem) were set to (256, 2, 512M) in TPC-W, (256, 1, 1.5G) in TPC-C, and (512, 2, 512M) in SPECweb. The settings optimized the overall performance for all the applications. Config-1 moves 1GB memory from TPC-C to SPECweb; Config-2 reduces the virtual CPU of TPC-W from 2



(a) Due to configuration



(b) Due to workload dynamics

Figure 5.1: Balance configurations is desirable for system-wide performance.

to 1; Config-3 moves 256 credits from SPECweb to TPC-C. Figure Figure 5.1(a) suggests that VM performance is sensitive to the process of resource allocation. In certain times, unexpected degraded performance is observed in a VM with even more resources. For example, in Config-1, with more memory the SPECweb VM had an unexpected worse performance. The reason is due to the increased competition for I/O bandwidth from the TPC-C VM which was de-allocated 1GB memory.

Figure Figure 5.1(b) plots the performance with fixed VM configurations while changing the load level in each application. The workload selections are defined in Table Table 5.1. By reducing the incoming workload to TPC-W, TPC-C and SPECweb, we got three workloads ordered from left to right in the figure. Intuitively, reduced traffic should result in better performance due to alleviated resource contention. However, the assertion does not hold in Figure Figure 5.1(b). In workload-1, reduced workload in TPC-W alleviated the CPU competition with SPECweb VM. However, with more chance to get scheduled, the SPECweb VM reduced the I/O bandwidth available to TPC-C which ended up with a performance loss. Although overall resource demand decrements, unbalanced VM configurations can still possibly lead to significant performance loss.

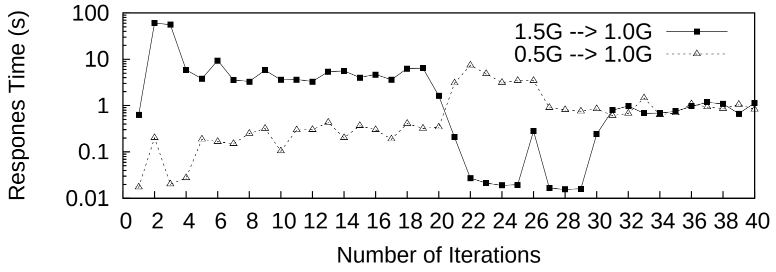


Figure 5.2: Delayed effect in memory reconfiguration.

5.2.3 Delayed Effects

VM capacity management relies on precise operations that set resources to desired values assuming the observation of the instant reconfiguration effect. Delayed effects are observed in both memory and CPU allocations. I/O data is transferred to and from each domain via Xen and the driver domain, using shared-memory. The hypervisor and driver domain may cache data to expedite VM I/O accesses. VMs with fewer memory may have more data cached by the VMM and driver domain. Thus the way of configuring a VM to its target memory size can potentially affect VMs' performance. Increasing or decreasing to the target memory size can have distinct effects. Due to the caching effects, the influence of a previous configuration may last several configuration steps. Figure Figure 5.2 shows the delayed effect in memory allocation. The target memory size for TPC-C benchmark was set to 1GB, but from initial settings of 1.5GB and 0.5GB. The memory size was adjusted (at the 20th minute) after the initial configuration produced stable response times. The effect of the adjustment lasted for several minutes before the response time stabilized again.

Similar phenomenon can also be observed in CPU allocation. We did tests measuring the dead time between a change in VCPU and the time the performance stabilizes. A single TPC-W DB tier was tested by changing its VCPU. Figure Figure 5.3 plots the application-level performance over time. Starting from 4 VCPUs, the VM was

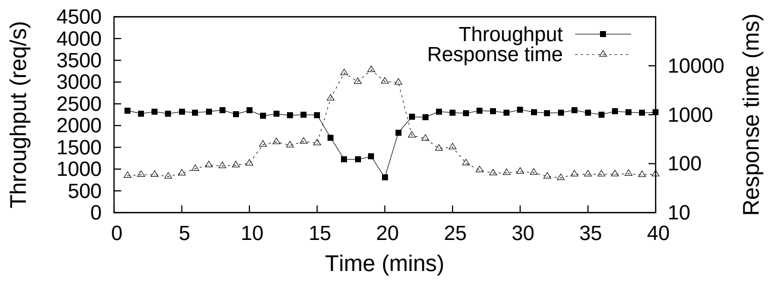


Figure 5.3: Delayed effect in VCPU reconfiguration.

removed one VCPU every 5 minutes until one was left at the time of the $15th$ minute. Then the VCPU was added back one by one. At the $20th$ minute, the number of VCPUs increased from 1 to 2. We observed a delay time of more than 5 minutes before the response time stabilized at the time of the $25th$ minute. The reason for the delay was due to the resource contention caused by the backlogged requests when there were more CPU available. The VM took a few minutes to digest the congested requests.

The complicated resource to performance relationship and possible delayed consequences of previous allocation decisions pose challenges to on-the-fly VM resource management. In server consolidation, the resource allocation may need to be changed due to workload dynamics. The system-wide performance (performance summarized over all hosting applications) should be optimized. This motivated us to design a VM configuration agent to automate the management process. The self-optimizing agent should be able to dynamically alter the VM settings to a better one in consideration of performance interference between VMs and the delayed effects. Reinforcement learning gives a possible solution to the problem.

5.3 Reinforcement Learning for VM Auto-configuration

5.3.1 Reinforcement Learning and Its Applicability to VM Auto-configuration

Reinforcement learning is concerned with how an agent ought to take actions in a dynamic environment so as to maximize long term rewards defined on a high level goal [67]. The RL offers two advantages [68]. First, it does not require a model of either the system in consideration or the environment dynamics. Second, RL is able to capture the delayed effect in a decision-making task.

The outcome of RL is a policy that maps the current state of the agent to the best action it could take. The "goodness" of an action in a state is measured by a value function which estimates the future accumulated rewards by taking this action. The RL-enabled agent performs trial-and-error interactions with the environment, each of which returns an instantaneous reward. The reward information is propagated backward temporally in repeated interactions, eventually leading to an approximation of the value function. The optimal policy is essentially choosing the action that maximizes the value function in each state. The interactions consist of exploitations and explorations. Exploitation is to follow the optimal policy; in contrast exploration is the selection of random actions to capture the change of the environment so as to enable the refinement of existing policy.

The VM configuration task fits within the agent environment framework. Consider the agent as a controller residing in `dom0`. The states are VM resource allocations; possible changes to the allocations form the set of actions. The environment comprises the dynamics underlying the virtualized platform. Each time the controller adjusts the VM configurations, it receives performance feedback from individual VMs. After sufficient interactions, the controller obtains good estimations of the "goodness" of the allocation decisions given current VM configurations. Starting from an arbitrary initial setting, the controller is able to lead the VMs to optimal configurations by following the optimal policy. Through explorations, the controller can modify its resource allocation policy according to the dynamics of VM traffics.

A RL problem is usually modeled as a *Markov Decision Process* (MDP). Formally, for a set of environment states $\mathcal{S}$ and a set of actions $\mathcal{A}$, the MDP is defined by the transition probability $P_a(s, s') = Pr(s_{t+1} = s'|s_t = s, a_t = a)$ and an immediate reward function $R = E[r_{t+1}|s_t = s, a_t = a, s_{t+1} = s']$. At each step t, the agent perceives its current state $s_t \in \mathcal{S}$ and the available action set $\mathcal{A}(s_t)$. By taking action $a_t \in \mathcal{A}(s_t)$, the agent transits to the next state s_{t+1} and receives an immediate reward r_{t+1} from the environment. The value function of taking action a in state s can be defined as:

$$Q(s, a) = E\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s, a_t = a\},$$

where $0 \leq \gamma < 1$ is a discount factor helping $Q(s, a)$'s convergence.

5.3.2 Formulation of VM Configuration as a RL Task

The VM configuration task is naturally formulated as a continuing discounted MDP. The goal is to optimize the overall VM(s) performance. We define the reward function based on individual VM's application level performance. The state spaces are the hardware configuration of VMs which are fully observable in the driver domain. Actions are the combination of the change to the configurable parameters. The configuration task is formulated as following:

The reward function. The long-term cumulative reward is the optimization target of RL. In the VM configuration task, the desired configurations are the ones which optimize system-wide performance. The immediate rewards are the summarized VM(s) performance feedbacks on the resulted new configuration. The performance of individual VM is measured by a score which is the ratio of current throughput (*thrpt*) to a reference throughput plus possible penalties when response time (*resp*) based SLAs (Service Level Agreement) are violated:

$$score = \frac{thrpt}{ref\_thrpt} - penalty.$$

$$penalty = \begin{cases} 0 & \text{if } resp \leq SLA; \\ \frac{resp}{SLA} & \text{if } resp > SLA. \end{cases}$$

The reference throughput (*ref_thrpt*) values are the maximum achievable application performance under SLA constraints in current hardware settings. We obtained the reference for one application by dedicating the physical host and giving more than enough resources to the corresponding VM. A low score indicates either lack of resource or interference between VMs, both of which should be avoided in making allocation decisions. Then, the immediate reward is the summarized scores over all VMs. As suggested by virtualization benchmarks [12, 77] for summarized performance, we define the reward as:

$$reward = \begin{cases} \sqrt[n]{\prod_{i=1}^{n} w_i * score_i} & \text{if for all } score_i > 0; \\ -1 & \text{otherwise,} \end{cases}$$

where w_i is the weight for the i^{th} VM, $1 \leq i \leq n$. We strictly refuse the configurations of negative scores (i.e. violation of SLA) by assigning a reward of -1. In the case of soft SLA thresholds, the reward function can be revised correspondingly to tolerate transient SLA violations.

The state space. In the VM configuration task, the state space is defined to be the set of possible VM configurations. In the driver domain, VM configurations are fully observable. States defined on the configurations are deterministic in that $P_a(s, s') = 1$, which simplifies the RL problem. We define the RL state as the global resource allocations:

$$(mem_1, time_1, vcpu_1, \cdots, mem_n, time_n, vcpu_n).$$

where mem_i, $time_i$ and $vcpu_i$ are the i^{th} VM's memory size, scheduler credit and virtual CPU number, respectively. Since the hardware resources available to VMs are limited, constraints exist. The value of mem should not exceed the total size of memory that can be allocated to VMs. In addition, $vcpu$ should be a positive integer, not exceeding the number of physical CPUs and the scheduler credit be positive, too.

The actions. For each of the three configurable parameters, possible operations can be either *increase*, *decrease* or *nop*. The actions for the RL task are defined as the combinations of the operations on each parameter. For parameters like $time$ and mem that are continuous, we quantify them by defining change steps. Memory is reconfigured in unit of 256MB; scheduler credit changes in a step of 256 credits and virtual CPU number is incremented or decremented by one at a time.

An action is invalid if by taking the action, the target state violates state constraints. Another restriction for taking action is that only one parameter is considered at a time and only one-step reconfiguration is allowed. It follows the natural trail-and-error method that searches the configuration state space exhaustively. More importantly, resource adjustment in small steps smooths the configuration process.

5.3.3 Solutions to the RL Task

The solution to a RL task is an optimal policy that maximizes the cumulative rewards at each state. It is equivalent to finding an estimation of $Q(s, a)$ which approximates its actual value. The experience-based solution is based on the theory that the average of the sample $Q(s, a)$ values collected approximates the actual value of $Q(s, a)$ given sufficiently large number of samples. A sample is in the form of (s_t, a_t, r_{t+1}). The basic RL algorithms in experience-based solution are called $temporal\text{-}difference$ (TD) methods, which update $Q(s, a)$ at each time a sample is collected:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha * [r_{t+1} + \gamma * Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)],$$

where α is the learning rate and γ is the discount factor. The Q values are usually stored in a look-up table and updated by writing new values to the corresponding entries in the table. In the VM configuration task, the RL-based agent issues reconfiguration actions following an ϵ-greedy policy. With a small probability ϵ, the agent picks a random action, and follows the best policy it has found for most of the time. Starting from any initial policy, the agent gradually refines the policy based on the feedback perceived at each step.

5.4 The Design and Implementation of VCONF

In this section, we introduce VCONF, a RL-based VM auto-configuration agent. Including multiple VMs in the RL problem poses challenges to the adaptability and scalability of VCONF. We address the challenges by employing model-based RL methods with two layers of approximation.

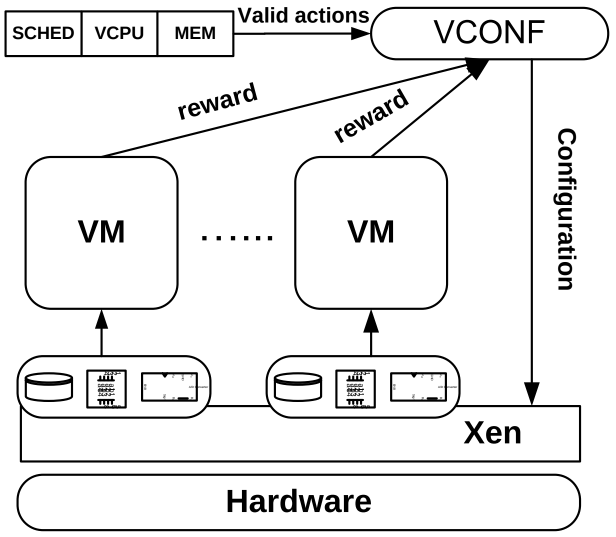


Figure 5.4: The organization of VCONF.

5.4.1 Overview

VCONF is designed as a standalone daemon residing in the driver domain. It takes advantage of the control interface provided by dom0 to control the configuration of individual VMs. Figure Figure 5.4 illustrates the organization of VCONF and the Xen virtualization environment. VCONF manages the VM configurations by monitoring performance feedbacks from each VM. Reconfiguration actions take place periodically based on a predefined time interval. VCONF queries the driver domain for current state and computes valid actions. Following the policy generated by the RL algorithm, VCONF selects an action and sends it to dom0 for VMs reconfiguration. At the end of each step, VCONF collects the performance feedbacks in each VM and calculates the immediate reward. The new sample of the immediate reward is processed by the RL algorithm and VCONF updates the configuration policy accordingly. VCONF implements a basic look-up table based Q function for small systems. In a larger system, VCONF employs model-based RL algorithm for adaptability and scalability.

5.4.2 Adaptability and Scalability

Adaptability is the ability of RL algorithms to revise the existing policy in response to the change of the environment. To adapt current policy to a new one, the RL agent needs to perform a certain amount of exploration actions, which are believed to be suboptimal actions leading to bad rewards. In production systems, the explorations can be prohibitively expensive due to bad client experiences. The RL algorithm usually requires a long time for new samples collection before a new policy can be derived. This is not acceptable for online policy generation tasks like VM auto-configuration.

Scalability issues refer to the problem that the number of Q values grows exponentially with the state variables. In a look-up table-based Q implementation, the values are stored separately without interactions. The convergence of the optimal policy depends critically on the assumption that each table entry be visited at least once. In practice, even if the storage and computation complexity for a large Q table are not a concern, the time required to collect sample rewards to populate the Q table is prohibitively long.

Instead of updating each $Q(s, a)$ value directly from the immediate reward recently collected, VCONF employs environment models to generate simulated experiences for value function estimation. The environment models are essentially data structures that capture the relationship between current configuration, action and the observed reward. The model can be trained from previous collected samples in the form of (s_t, a_t, r_{t+1}) using supervised learning. Once trained, a model is able to predict the r values for unseen state-action pairs.

The use of environment models offers two advantages for RL tasks: First, model-based RL is more data efficient [7]. With limited samples, the model is able to shed insight on unobserved rewards. Especially in online policy adaptation, the model is updated every time with new collected samples. The modified model generates simulated experiences to update the value function, and hence expedites policy adaptation. Second, the immediate reward models can be reused in a similar environment. The environmental dynamics in VM configuration task are the time-varying resource demands in each VM. Different models can be learned for different combination of demands in VMs. We call such a combination a workload. In online adaptation, once VCONF identifies the resource demand is similar to a previous workload, the corresponding model is re-used. Instead of starting from scratch, the reuse of previous models is equivalent to starting from guided domain knowledge, which again improves online performance.

In model-based RL, the scalability problem is alleviated by the model's ability in coping with relatively scarcity of data in large scale problems. The conventional table-based Q values can be updated using the batch of experiences generated by the environment model. However, the table-based Q representation requires a full population using the rewards simulated by the model. This is problematic when the RL problem scales up. In VCONF, we use another layer of approximation for the value function, which helps to reduce the time in updating the value function in each configuration step.

5.4.3 Model Initialization and Adaptation

We selected standard multi-layer feed-forward back propagation neural network (NN) with sigmoid activations and linear output to represent the environment model. The selection was due to NN's ability to generalise from linear to non-linear relationship between the environment and the real-valued immediate reward. More impor-

Algorithm 1 The VCONF online algorithm

```
1: Initialize $Q_{appx}$ to trained function approximator.
2: Initialize $t \leftarrow 0, a_t \leftarrow nop$.
3: repeat
4:     $s_t \leftarrow get\_current\_state()$
5:     $re\_configure(a_t)$
6:     $r_{t+1} \leftarrow observe\_reward()$
7:     $a_{t+1} \leftarrow get\_next\_action(s_t, Q_{appx})$
8:     $worload \leftarrow identify\_workload()$
9:     $R_{model} \leftarrow select\_model(workload)$
10:    $update\_R_{model}(s_t, a_t, r_{t+1}, R_{model})$
11:    $update\_Q_{appx}(R_{model}, Q_{appx})$
12:    $t \leftarrow t + 1$
13: until VCONF is terminated
```

tantly, it is easy to control the structure and complexity of the network by changing the number of hidden layers and the number of neurons in each layer. This flexibility facilitates the integration of supervised learning algorithms with RL for better convergence. The performance of model-based RL algorithms depends on the accuracy of the environment model in generating simulated samples. Thus, the training samples used to train the model should be representative. We generated the training samples for the model by enumerating important configurations. In the implementation of Q function, an NN-based function approximator replaces the tabular form. The NN function approximator takes the state-action pairs as input and outputs the approximated Q value. It directs the selection of reconfiguration actions based on the ϵ-greedy policy.

Algorithm 1 shows the VCONF online algorithm. VCONF is designed to run forever until being stopped. At each configuration interval, VCONF records the previous state and observes the actual immediate reward obtained. Next action is selected by ϵ-greedy policy according to output of function approximator Q. VCONF identifies the workload by examining system-level metrics during last interval. The function *select_workload* is implemented in a way similar to the one in [59] using

Algorithm 2 Update the Q approximator

```
1: Initialize $Q_{appx}$ to the current function approximator.
2: repeat
3:     $sse \leftarrow 0$
4:     for $n$ iterations do
5:         $(s_t, a_t, r_t) \leftarrow generate\_sample(R_{model})$
6:         $target \leftarrow r_t + \gamma * Q_{appx}(s_{t+1}, a_{t+1})$
7:         $error \leftarrow target - Q_{appx}(s_t, a_t)$
8:         $sse \leftarrow 0.9 * sse + 0.1 * error * error$
9:         train $Q_{appx}(s_t, a_t)$ towards $target$
10:    end for
11: until $converge(sse)$
```

supervised learning except that the output is the predicted workload type. The new sample (s_t, a_t, r_{t+1}) then updates the selected environmental model. The Q function approximator is batch-updated as in Algorithm 2.

5.5 Experimental Results

5.5.1 Methodology

We designed a set of experiments to show the effectiveness of the RL-enabled VCONF in VM auto-configuration. Figure Figure 5.5 lists four different VM settings. The experiments are divided into two parts. In the first part, VCONF was evaluated in controlled environments in which the number of applications and resources were limited to a small set. The multi-tier TPC-W benchmark was selected as the application. As in Figure Figure 5.5(a), its reference performance was obtained by running TPC-W application server and database server on two separate physical servers exclusively. Figure Figure 5.5(b) shows a single instance of TPC-W with two tiers. Since TPC-W is primary CPU-intensive, the memory parameter was fixed in this controlled environment. By adjusting CPU resources allocated to each tier, VCONF is to max-

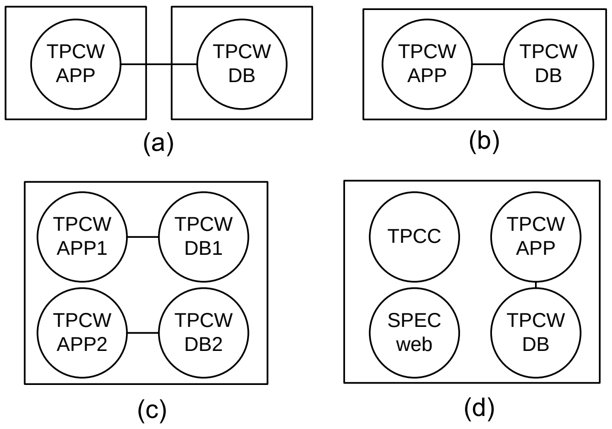


Figure 5.5: The design of experiments.

imize TPC-W's throughput. The experiment in Figure Figure 5.5(c) augmented the single application problem by adding another instance of TPC-W. VCONF needs to optimize system-wide performance finding balanced CPU allocation schemes for competing applications. In the second part, restrictions on the number of applications and resources in consideration were relaxed. As in Figure Figure 5.5(d), three applications with heterogeneous resource demands were consolidated in the host. The memory parameter needs to be considered. In the scaled-up problem, the state-action space is considerably larger than the controlled experiments. VCONF's implementation of model-based RL algorithm was evaluated and compared with basic RL methods.

5.5.2 Experiment Settings

The machines used in the experiments consist of virtual servers, client and compute machines. The physical machines for virtual hosting are Dell PowerEdge1950 with two quad-core Intel Xeon CPU and 8GB memory. In the controlled experiment, all VMs were pinned to the first four cores. We separated the RL related computation to a compute node in order to avoid possible VM performance interference. All the client and compute nodes were the same model Dell machines and were connected by

Gigabyte Ethernet network.

We used Xen version 3.1 as our virtualization environment. Both dom0 and the guest VMs were running CentOS Linux 5.0 with kernel 2.6.18. The VMs mounted their file-based disk images through a NFS server. For the benchmark applications, MySQL, Tomcat and Apache were used for database, application and web servers. The VM configuration actions were issued through dom0's privileged control interface `xm`.

We selected the TPC-W [73], TPC-C [74] and SPECweb [71] benchmarks as the workloads running within the VMs. They are typical server applications in today's data centers which are the targets of virtualization technology.

5.5.3 Applicability of RL-based VM Autoconfiguration

First, we studied the applicability of RL algorithms in the VM configuration task. In [65], the authors assumed independence of configuration parameters. With this assumption, VM configuration task can be easily solved by greedy search in each resource dimension. They showed database query costs drop linearly with more CPU shares. The cost is independent with the memory size allocated to the VM. Thus, greedy search together with linear regression are sufficient to find the optimal configuration without visiting every possible configurations. However, the independence assumption does not always hold. Due to the involvement of dom0 in VM execution, applications hungry for memory can be affected by CPU-intensive applications. Figure Figure 5.6 plots the performance of TPC-C under different CPU settings: equal, more and less. The VM competing for resource is an instance of TPC-W. "equal", "more" and "less" indicate 50%, 80% and 20% CPU allocations for TPC-C, respectively. The figure suggests a strong correlation between memory and CPU in deter-

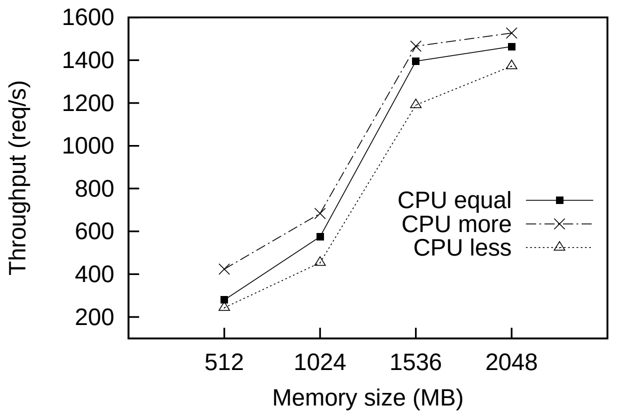


Figure 5.6: TPC-C performance in different settings.

mining application performance. That is, regression based greedy search approach needs to search the entire configuration space.

The RL algorithm does not assume any model of the system in consideration. It derives policies from interactions and continues to refine the policy with newly collected experiences. We validated the effectiveness of RL methods in VM auto-configuration starting from a simple problem. As showed in Figure Figure 5.5(b), a two-tier TPC-W application was hosted by the virtual server. We assume the application throughput as the optimization target. Requests execution in TPC-W involves processing on both tiers. Thus, the resulted performance is affected by the processing capacity on both tiers. TPC-W defines three different traffic mixes: shopping, browsing and ordering mix. Different traffic mixes put processing pressure on distinct tiers. Thus, it is not easy to determine the CPU assignment to each tier for balanced configuration. Moreover, due to dynamic CPU demands from different traffic mixes, existing CPU allocation needs to be frequently revised.

To limit the problem size, we restricted each tier to have up to 3 virtual CPUs. Only three scheduler credit assignments were selected: equal share, tomcat tier with 80% share and Database tier with 80% share. The resulted state space contains 27

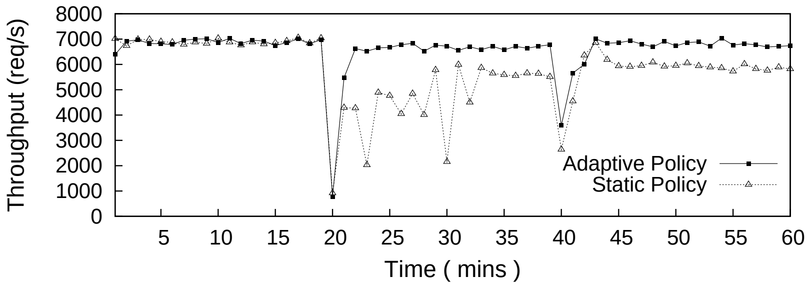


Figure 5.7: VCONF performance with TPC-W application.

configurations. VCONF was deployed with a table-based Q function which was initialized to all zeros. We used the $Sarsa(0)$ algorithm with $\alpha = 0.1, \gamma = 0.9, \epsilon = 0.1$ to drive the configuration agent, the configuration interval was set to 60 seconds. If otherwise specified, the same RL parameters and interval were used in the remaining experiments. The agent exits until the Q function converges. An optimal configuration policy can then be derived from the Q table. The RL learning process was repeated for above three workloads resulting in three policies.

Figure Figure 5.7 shows the online performance of VCONF with adaptive and static policies. The plots are the achieved throughput in TPC-W. During the testing, workloads were dynamically switched in the order of ordering, shopping and browsing mix every 20 minutes. VCONF with adaptive policies continuous monitored the system level performance metrics and identified workload changes. The policies were switched accordingly as recommended by VCONF. Configured with a static initialization policy, VCONF revised the initial policy only based on online interactions. The figure suggests that both RL agents were able to automatically drive an inappropriate configuration to a better setting in a small number of steps. The TPC-W throughput was brought up and maintained at a high level. The adaptive agent achieved the optimal performance, which is the best possible result for the RL approach. The one with the static policy also showed the effectiveness of RL, but with limited adaptability to

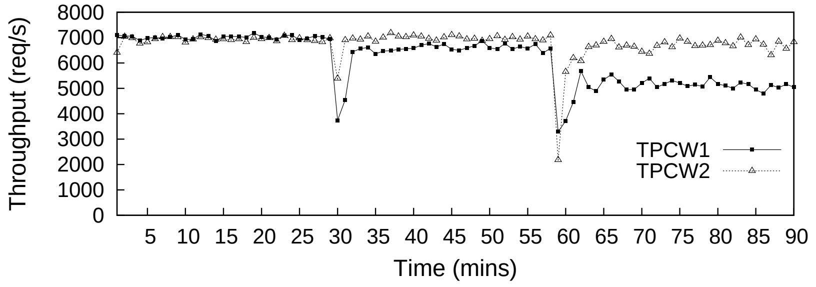


Figure 5.8: VCONF performance with two TPC-W instances.

a new policy when traffic changed.

5.5.4 RL-based System-wide Performance Optimization

In this experiment, we add one more TPC-W application to the problem. The goal of the RL agent is to maximize the cumulative reward which is defined as the summarized performance scores over both TPC-W instances. Adding more applications complicates the VM configuration problem. As the state space grows, the time required for the RL agent to obtain an optimal policy in online interaction becomes prohibitively long. For example, in the case of two TPC-W instances with two application server VMs and two database VMs, the state space increases to around 400 states if the state is defined similarly as in the first experiment. It would take the agent more than 400 minutes to visit every state. The convergence of the Q function usually requires multiple visits to each entry and the RL agent following the $\epsilon-greedy$ policy may not update different entries each time. Thus, the resulted time required for online RL learning is unacceptable. One possible solution is to pre-define a policy that guides the RL agent in online learning. Upon an acceptable good policy is derived from online guided interactions, the RL agent is handed over to the generated policy.

We designed the initial policy to be as simple as visiting different configurations at each step. As more states are visited, the RL agent performs sweeps of batch updates to the Q table using the collected rewards. In this experiment, the pre-defined policy terminates when all configurations have been visited. Due to the presence of delayed effects, different sequences of visiting may receive different rewards. Theoretically, the Q function approximates its actual value only if the agent perceives the effect of all the state-action pairs. Thus, the generated policy still needs online refinement before the optimal policy is achieved. In practice, near optimal policy often satisfies users' requirement.

The RL-based VM resource management is to optimize both applications in operation. Because VMs with identical resource demands can be configured to have the same resource allocation. To test VCONF, the hosted TPC-W applications ran different traffic mixes. Randomly selecting two traffic mixes as the input traffic to the VMs forms three different resource demands for the whole system. The optimization goal for the RL agent is to maximize system wide throughput for both applications. Figure Figure 5.8 shows the change of their throughput during RL online learning. The incoming workload changes every 30 minutes. We randomly selected a time period with three different workloads and evaluated VCONF's ability in system wide performance optimization. For TPC-W1, the traffic mix changes were: ordering, ordering and shopping. To form different resource demands, TPC-W2 ran shopping, browsing and browsing mixes correspondingly. From the figure, we can see that both applications suffered performance degradation when the workload changed at the $30th$ and $60th$ time points. This is partially due to unbalanced VM configuration caused by traffic dynamics. On the other hand, the RL agent was able to correct unbalanced configurations within a few steps. For example, TPC-W1's throughput dropped to 2000 during the second workload change. The RL agent brought the performance back and maintained the throughput around 7000 within 7 steps. Note that the pol-

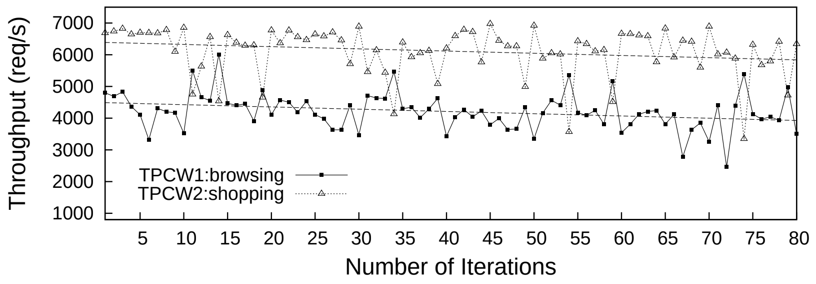


Figure 5.9: Performance of trial-and-error.

icy employed by the RL agent is not guaranteed to be an optimal policy because of the agent's limited interactions with the environment. There is no guarantee that the throughputs for both applications were maximized.

From the $60th$ time point, the two VMs ran browsing and shopping mixes respectively. The resource contention and performance interference between the two VMs are more pronounced under this workload. We examined the effectiveness of RL-based approach by comparing the performance of the derived RL policy with a general trial-and-error method. The method fixes the value of one parameter and tries different settings for another parameter. Figure Figure 5.9 plots performance of the trial-and-error method. The trend line in the figure is the linear regression of the performance in both VMs. The figure suggests that, on average the VMs running browsing and shopping mix can achieve a maximum throughput of 4500 and 6500 concurrent requests. Compared with the trial-and-error, the RL-based approach brought the throughput of both applications to around 5000 and 7000 respectively. More importantly, the RL approach automatically directed the resource allocation towards target configurations without any human intervention.

We define the difference between the system throughput under current configuration and the throughput achieved in the target configuration as the performance deviation. Figure Figure 5.10 plots the performance deviation in each configuration step with a 95% confidence interval. The figure suggests that starting from arbitrary configurations, the RL agent should be able to continuously improve the system throughput at each configuration step. On average, the system wide throughput would stabilize within 7 configuration steps.

5.5.5 Model-based RL in VM Auto-configuration

In previous experiments, we showed the effectiveness of RL in small scale problems. VCONF was able to find the optimal configuration for a single application. In the multiple-application problem, a policy generated by RL using previous collected traces achieved good results in optimizing system wide performance. Statistical results showed that the RL approach would continuously improve the configuration step by step and reach the target configuration within a small number of iterations. However, as the VM configuration problem scales up, the state space grows dramatically.

Standard RL approaches depends critically on the experiences with the environment to generate policies. Unfortunately, the number of experiences needed for an optimal RL policy grows with the state space. The pre-defined policy used to collect experiences is likely to converge to sub-optimal policies due to the relatively data scarcity in the huge state space. Model-based RL provides a solution to the problem by providing a generalization over the collected experiences. By training a model that captures the relationship between state-action pairs and the rewards collected, the RL agent is able to simulate experiences for unseen state-action pairs. Then, the simulated experiences are used to update the Q values. The performance of the model-based RL approach relies on the accuracy of the trained model. Policies for experience collection should be carefully designed in order to record representative sample data.

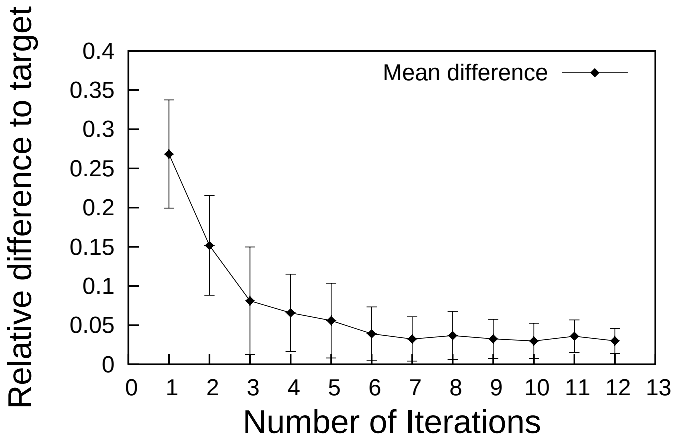


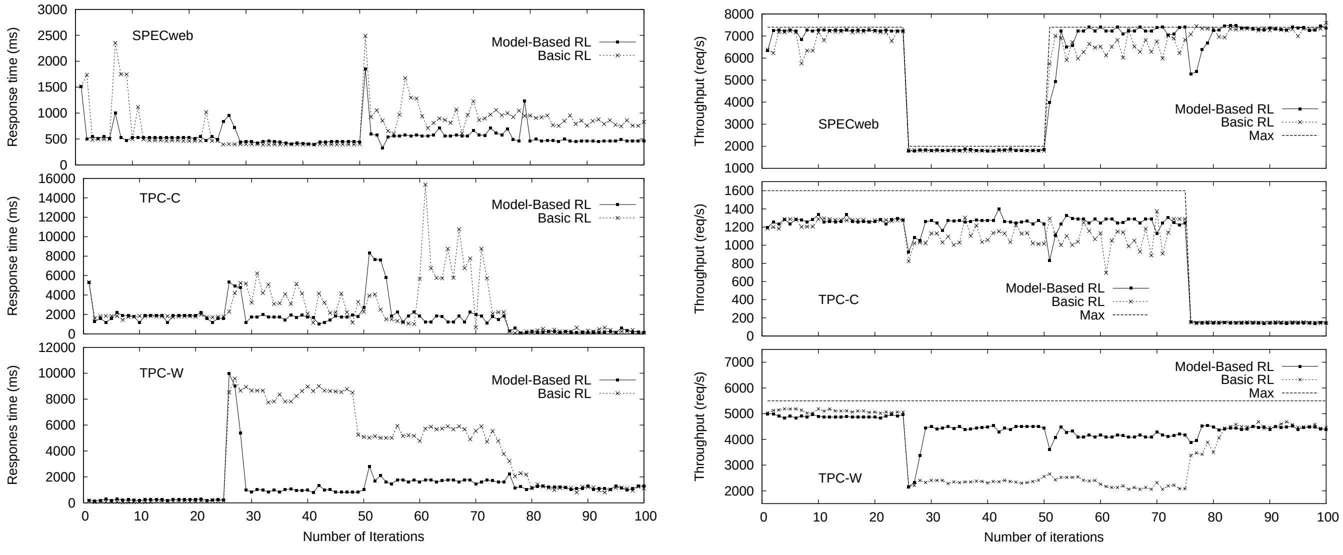
Figure 5.10: Performance deviation during re-configuration.

In the last experiment, we scaled the previous controlled VM configuration problem in two dimensions. We consolidated three benchmarks, TPC-W, TPC-C and SPECweb, with heterogeneous resource profiles in the virtual server. TPC-W is primary CPU-intensive while TPC-C requires a large amount of disk I/Os. The execution of requests in SPECweb involves processor and network I/O for dynamic content generation and static image serving. The VM resources in consideration were the virtual CPU number, scheduler credit and memory size. We defined different workloads with varying resource demands and tabularized them in Table Table 5.1.

All VMs were initially set to an identical configuration: 1.5GB memory, 4 virtual CPUs and a credit of 256. We designed the policy for experience collection as a traversal in a pre-defined resource configuration set. The set contains representative combinations of the allocations uniformly scattered in the state space. The NN models were trained with a learning rate of 0.0001 and a momentum of 0.1. Four models were trained for different workloads. A second layer NN generalization was used as the Q function approximator and its learning process is listed in Algorithm 2. The time required to train a NN model from an arbitrary neural network is approximately 10 minutes. When updated incrementally, the training time reduces to around 1 minute.

Table 5.1: Workload settings.

	TPC-W	TPC-C	SPECweb
workload-0	600 browsing clients	50 warehouses, 10 terminals	800 banking clients
workload-1	600 ordering clients	50 warehouses, 10 terminals	800 banking clients
workload-2	600 browsing clients	50 warehouses, 1 terminal	800 banking clients
workload-3	600 browsing clients	50 warehouses, 10 terminals	200 banking clients



(a) Response Time (b) Throughput

Figure 5.11: Performance of VCONF with heterogeneous applications.

In order to fit the updates of the NN model and the Q approximator between each interval, we limited the update of the NN model and the Q approximator to 50 iterations and 100 sweeps respectively, which resulted in a 50-second compute time. We compared model-based RL approach with the basic table-base RL algorithm. To be fair comparison, the basic RL's Q tables for different workloads were initialized by the NN-based Q approximators. During online learning, VCONF identifies workload changes and recommends corresponding models and Q tables to model-based RL agent and basic RL agent. Both the model-based RL agent and the basic RL agent were started with the same VM initial configuration.

We randomly selected a time period with four different workloads. Figure Figure 5.11 shows the performance of VCONF with respect to response time and throughput. The "Max" plot is the reference throughput for each application. The reference value was obtained when each application ran alone on the virtual server with sufficient resources. Due to VM interferences and possible inappropriate configurations, the throughput for each application is less than the reference value. Model-based RL approach outperforms basic RL in that it achieved a higher throughput and lower response time during online learning. The model-based RL was able to adapt to workload changes well. It improved the application throughput by 20%-100% over the basic RL approach in different applications. In addition, model-based approach was more stable sticking with the "best" configuration during the same workload. The basic RL agent wagered between several configurations some of which incurs considerable performance penalty.

The advantage of model-based RL approach over basic RL is due to the model's ability generalizing the environmental changes. In another word, the model-based approach is more data efficient [7] that a change in the environment can spread to other state-action pairs because they are co-related within the model. The basic RL approach stores Q values separately without interactions, then an environmental change can only influence the agent's decision when the affected Q value entry is visited next time.

5.6 Summary

In this work, we present VCONF, a RL-based agent for virtual machine auto-configuration. VCONF automates the VM reconfiguration process by generating policies learned from iterations with the environment. Experiments on Xen VMs with typical server applications showed VCONF's optimality in controlled problems and good adaptability and scalability in a cloud computing testbed. In the presence of workload dynamics, VCONF was able to adapt to a good configuration within 7 steps and showed 20% to 100% throughput improvement over basic RL methods. Although, there is no optimality guarantee for the derived configurations, VCONF was able to direct arbitrary initial configuration to a better one without performance penalties in any of the VMs.

Chapter 6

Resource Management in Virtual Clusters

6.1 Introduction

Exporting infrastructure as a service gives cloud users the flexibility to select VM operating systems (OS) and the hosted applications. But this poses new challenges to underlaying VM management as well. Because public IaaS providers assume no knowledge of the hosted applications, VM clusters of different users may overlap on physical servers. The overall VM deployment can show an dependent topology with respect to resources on physical hosts. The bottleneck of multi-tier applications can shift between tiers either due to workload dynamics or mis-configurations on one tier. Mis-configured VMs can possibly become rogue ones affecting others. In the worst case, all nodes in the cloud may be correlated and mistakes in the capacity management of one VM may spread onto the entire cloud.

Our work in Chapter 5 demonstrates the efficacy of reinforcement learning (RL)-based resource allocation in a static cloud environment that VMs are deployed on one physical machine. Based on state space defined on co-running VM configurations, we optimize system-wide VM performance on one machine under different workload combinations. Although effectively managing configurations of VMs with distinct resource demands, the approach in Chapter 5 assumes a static environment and relies on workload specific environment models to map VM configurations to system-wide performance index. This approach can not be easily extended to a dynamic cloud environment, in which VMs are hosted on a cluster of physical machines. First, it becomes prohibitively expensive to maintain models for different workload combinations as the number of VMs increases. Second, possible VM join/leave or migration makes the optimization of cluster-wide performance difficult. Finally, the state space defined on VM configurations is not robust to workload dynamics.

In this chapter, we address the issues and present a distributed learning approach for cloud management. We decompose the cluster-wide cloud management problem into sub-problems concerning individual VM resource allocations and consider cluster-wide performance to be optimized if individual VMs meet their SLAs with a high resource utilization. To handle workload dynamics, we extend the state definition in Chapter 5 from VM configurations to VM running status and address the issues due to the use of continuous running status as the state space. More specifically, our contributions are as follows:

(1) Distributed learning mechanism. We treat VM resource allocation as a distributed learning task. Instead of cloud resource providers, cloud users manage individual VM capacity and submit resource requests based on application demands. The host agent evaluates the aggregated requests on one machine and gives feedback to individual VMs. Based on the feedbacks, each VM learns its capacity management policy accordingly. The distributed approach is scalable because the complexity of the management is not affected by the number of VMs and we rely on implicit coordination between VMs belonging to the same virtual cluster.

(2) Self-adaptive capacity management We develop an efficient reinforcement learning approach for the management of individual VM capacity. The learning agent operates on a VM's running status which is defined on the utilization of multiple resources. We employ a Cerebellar Model Articulation Controller-based Q table for continuous state representation. The resulted RL approach is robust to workload changes because state on low-level statistics accommodate workload dynamics to a certain extent.

(3) Resource efficiency metric. We explicit optimize resource efficiency by introducing a metric to measure a VM's capacity settings. The metric synthesizes application performance and resource utilization. When employed as feedbacks , it effectively punishes decisions that violate applications' SLA and gives users incentives to release unused resources.

(4) Design and implementation of iBalloon. Our prototype implementation of the distributed learning mechanism, namely iBalloon, demonstrated its effectiveness in a Xen-based cloud testbed. iBalloon was able to find near optimal configurations for a total number of 128 VMs on a 16-node closely correlated cluster with no more than 5% of performance overhead . We note that, there were reports in literature about the automatic configuration of multiple VMs in a cluster of machines. This is the first work that scales the auto-configuration of VMs to a cluster of correlated nodes under work-conserving mode.

6.2 Motivating Examples

In this section, we first review the complications of CPU, memory and I/O resource allocations in a cloud. These complications motivated us to develop a resource management scheme that works directly on the actual resource usage instead of nominal allocations. Second, we give an example showing that virtual cluster applications such as multi-tier websites can possibly create dependencies across nodes in a cluster, which makes the optimization of resource allocation in a cluster difficult.

6.2.1 Complications in Multiple Resource Allocation

In cloud computing, application performance depends on the application's ability to simultaneously access multiple types of resources. In this work, we consider CPU, memory and I/O bandwidth as the building blocks of a VM's capacity. An accurate resource to performance model is crucial to the design of automatic capacity management. However, the workload and cloud dynamics make the determination of the system model challenging. Our discussions are based on Xen virtualization platforms, but they are applicable to other virtualization platforms like VMware and VirtualBox. In the Xen based platform, the driver domain (`dom0`) is a privileged VM residing in the host OS. It manages other guest VMs (`domU`) and performs the resource allocations. In the rest of this chapter, we use `dom0` and the host interchangeably. VMs always refer to the guest VMs or `domUs`.

CPU

The CPU(s) can be time-shared by multiple VMs in fine-grain. For example, the Credit Scheduler, which is the default CPU scheduler in Xen, can perform the CPU allocation in a granularity of 30 ms. On boot, each resident VM is assigned a certain number of virtual CPU (VCPU), and the number can be changed on-the-fly. Although the number of VCPUs does not determine the actual allocation of CPU cycles, it decides the maximum concurrency and CPU time the VM can achieve. In general, CPU scheduling works in a *work-conserving (WC)* or *non-work-conserving (NWC)* mode.

It is convenient to obtain the VMs' CPU utilization. The usage can be reported by `dom0` using `xentop` or by the VM's OS (e.g. the `top` command in linux). However, it is easily to determine how CPU resources are allocated to VMs. In general, there are three ways of CPU allocation:

1. Under WC mode, set VMs' VCPU to the number of available physical CPU and change the CPU allocations by altering VMs priorities (or *weight* in Xen).

2. Under WC mode, change CPU allocation by altering the VCPU number. It is equal to setting an upper limit of CPU allocation to the VCPU number. Within the limit, a VM can use CPU for free.

3. Under NWC mode, same as the first method, except that the allocations are specified as cap values. All the cap values add up to the total available CPU resource.

To determine the best CPU mode in cloud management, we compared the above three methods on a host machine with two quad-core Intel Xeon CPUs. Two instances of TPC-W database (DB) tier were consolidated on the host. For more details about the TPC-W application, please refer to Section 6.5. The DB tier is primary CPU-intensive and the VMs were limited to use the first four cores only. We make sure that the aggregated CPU demand is beyond the total capacity of four cores.

Figure Figure 6.1 draws the aggregated throughput and average response time of two TPC-W instances, under different CPU allocation modes. WC-4VCPU refers to the first method with equal weight of the two VMs. Although the aggregated CPU demand is beyond four cores, each VM actually needs a little more than two cores. It is equivalent to work-conserving with "over-provisioning" of CPU to individual VMs. WC-2VCPU is similar except that there is a 2-VCPU upper limit for each VM. In NWC-capped, we set the VMs to have 4 VCPU and each of the VM was capped to half of the CPU time. For example, in the case of four cores, a cap of 400 means no limit while 200 refers to half of the capacity.

In the figure, we can see that WC-2VCPU provided the best performance in terms of both throughput and response time. Plausible reasons for the compromised performance in the other two modes can be attributed to possible wasted CPU time. CPU contentions in WC-4VCPU may lower the CPU efficiency in serving requests. In principle, NWC-capped should deliver similar performance as WC-2VCPU. In practice, the results due to WC-2VCPU turned out to be better than those of NWC-capped.

Under NWC mode, there is usually a simple (and often linear) relationship between CPU resource and application performance. In [51], the authors showed an auto-regressive-moving-average model can represent this relationship well. However, in WC mode, the actual allocated CPU time to a VM is determined by the total CPU demand on the host, which makes the modeling harder. We take the challenges to consider WC mode in the VMs capacity management because it provides better performance and avoids possible waste of CPU resource.

Memory

Unlike CPU, memory is usually shared by dividing the physical address space into non-overlapping regions, each of which is used dedicatedly by one VM. Although it is possible for a VM to give up unused memory through self-ballooning [46], during each management interval we consider the allocated memory be used exclusively by one VM. The objective of the cloud memory management is to dynamically balancing

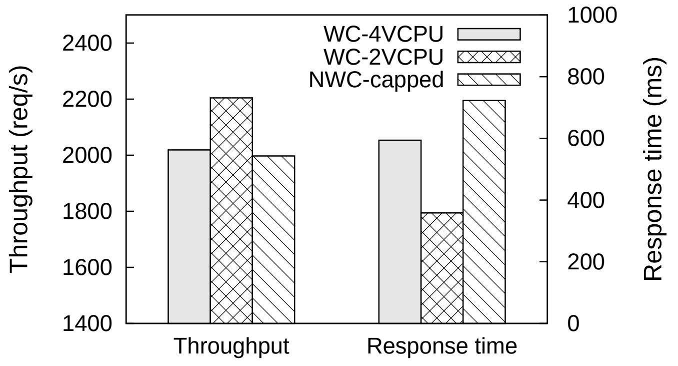


Figure 6.1: Performance of TPC-W under different CPU allocation modes.

"unused" memory from idle VMs to the busy ones. Identification of "unused" memory pages or calculation of the memory utilization of a running VM is not trivial. Different from free pages, "unused" pages refer to those that once touched but not actively being accessed by the system. It can be calculated as the total memory minus the system working set.

System working set size (WSS) can be estimated either by monitoring the disk I/O and major page faults [35], or using miss ratio curve [92]. But these methods are only sensitive to memory pressure and are able to increase VM memory size accordingly. Any decrease of memory usage can not be quickly detected. As a result, the memory of a VM may not be shrunk promptly.

In concept, the relationship between VM memory size and application-level performance is simple. That is, the performance drops dramatically when the memory size is smaller than the application's WSS. The open cloud environment adds one more uncertainty to VM memory management. Modern OSes usually design their write-back policies based on system wide memory statistics. For example, in Linux, by default the write-back is triggered when 10% of the total memory is dirty. A change of VM memory size may trigger background write-backs affecting application performance considerably although the new memory size is well above the WSS.

I/O Bandwidth

All the I/O requests from VMs are serviced by the host's I/O system. If the host's I/O scheduler is selected properly, e.g. the CFQ scheduler in Linux, VMs can have differentiated I/O services. Setting a VM to a higher priority leads to higher I/O bandwidth or lower latency. The achieved I/O performance depends heavily on the sequentiality of the co-hosted I/O streams as well as their request sizes. Thus, the I/O usage, e.g. the achieved I/O bandwidth reported by command like `iostat`, does not directly connect to application performance.

There are two key impediments in mapping the memory or I/O resources to application performance. First, it is difficult to accurately measure the utilization of the resources. Second, the actual resource allocation (e.g. achieved I/O bandwidth) is determined by the characteristics of the applications as well as the co-running VMs.

6.2.2 Cluster-wide Correlation

In a public cloud, multi-tier applications spanning multiple physical hosts require all tiers to be configured appropriately. In most multi-tier applications, request processing involves several stages at different tiers. These stages are usually synchronous in the sense that one stage is blocked until the completion of other stages on other tiers. Thus, the change of the capacity of one tier may affect the resource requirement on other tiers. In Table Table 6.1, we list the resource usage on the front-end application tier of TPC-W as the CPU capacity of the back-end tier changed. APP MEM refers to the minimum memory size that prevents the application server from doing significant swapping I/Os; APP CPU% denotes the measured CPU utilization. The table suggests that, as the capacity of the back-end tier increases, the demand for memory and CPU in the front tier decreases considerably. An explanation is that

Table 6.1: Configuration dependencies of multi-tier VMs.

DB VCPU	1VCPU	2VCPU	3VCPU	4VCPU
APP MEM	790MB	600MB	320MB	290MB
APP CPU%	61%	47%	15%	10%

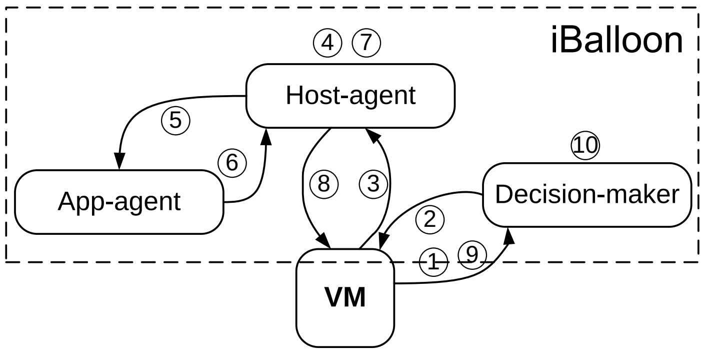


Figure 6.2: The architecture and working flow of iBalloon.

without prompt completion of requests at the back-end tier, the front tier needs to spend resources for unfinished requests. Therefore, any mistake in one VM's capacity management may spread to other hosts. In the worst case, all nodes in cloud could be correlated by multi-tier applications.

6.3 The Design of iBalloon

6.3.1 Overview

We design iBalloon as a distributed management framework, in which individual VMs initialize the capacity management. iBalloon provides the hosted VMs with capacity directions as well as evaluative feedbacks. Once a VM is registered, iBalloon maintains its application profile and history records that can be analyzed for future capacity management. For better portability and scalability, we decouple the functionality of iBalloon into three components: `Host-agent`, `App-agent` and

`Decision-maker`.

Figure Figure 6.2 illustrates the architecture of iBalloon as well as its interactions with a VM. `Host-agent`, one per physical machine, is responsible for allocating the host's hardware resource to VMs and gives feedback. `App-agent` maintains application SLA profiles and reports run-time application performance. `Decision-maker` hosts a learning agent for each VM for automatic capacity management. We make two assumptions on the self-adaptive VM capacity management. First, capacity decisions are made based on VM running status. Second, a VM relies on the feedback signals, which evaluates previous capacity management decisions, to revise the policy currently employed by its learning agent.

The assumptions together define the VM capacity management task as an autonomous learning process in an interactive environment. The framework is general in the sense that various learning algorithms can be incorporated. Although the efficacy or the efficiency of the capacity management may be compromised, the complexity of the management task does not grow exponentially with the number of VMs or the number of resources. After a VM submits its SLA profile to `App-agent` and registers with `Host-agent` and `Decision-maker`, iBalloon works as follows: (each step corresponds to a numbered interaction in Figure Figure 6.2)

① The VM reports its running status.

② `Decision-maker` replies with a capacity suggestion.

③ The VM submits the corresponding resource request to `Host-agent`.

④ `Host-agent` synchronously collect all VMs' requests, reconfigures VM resources and sleeps for a management interval.

⑤ `Host-agent` queries `App-agent` for the VM's application-level performance.

⑥ `App-agent` reports application-level performance.

⑦ Based on the application performance, Host-agent calculates the feedback accordingly.

⑧ `Host-agent` sends the feedback to the VM.

⑨ The VM wraps the information about this interaction and reports it to `Decision-maker`.

⑩ `Decision-maker` updates the capacity management policy for this VM accordingly.

iBalloon considers the VM capacity to be multidimensional, including CPU, memory and I/O bandwidth. This is one of the earliest works that consider these three types of resources together. A VM's capacity can be changed by altering the VCPU number, memory size and I/O bandwidth. The management operation to one VM is defined as the combination of three meta operations on each resource: *increase*, *decrease* and *nop*.

6.3.2 Key Designs

VM Running Status

VM running status has a direct impact on management decisions. A running status should provide insights into the resource usage of the VM, from which constrained or over-provisioned resource can be inferred. We define the VM running status as a vector of four tuples.

$$(u_{cpu}, u_{io}, u_{mem}, u_{swap}),$$

where u_{cpu}, u_{io}, u_{mem}, u_{swap} denote the utilization of CPU, I/O, memory and disk swap, respectively. As discussed above, memory utilization can not be trivially determined. We turn to guest OS reported metric to calculate u_{mem}(See Section 6.4 for details). Since disk swapping activities are closely related to memory usage, adding u_{swap} to the running status provides insights into memory idleness and pressure.

Feedback Signal

The feedback signal ought to explicitly punish resource allocations that lead to degraded application performance, and meanwhile encouraging a free-up of unused capacity. It also acts as an arbiter when resource are contented. We define a real-valued *reward* as the feedback. Whenever there is a conflict in the aggregated resource demand, e.g. the available memory becomes less than the total requested memory, iBalloon set the reward to -1 (penalty) for the VMs that require an increase in the resource and a reward of 0 (neural) to other VMs. In this way, some of the conflicted VMs may back-off leading to contention relaxation. Note that, although conflicted VMs may give up previous requests, `Decision-maker` will suggest a second best plan, which may be the best solution to the resource contention.

When there is no conflict on resources, the reward directly reflects application performance and resource efficiency. We define the reward as a ratio of *yield* to *cost*:

$$reward = \frac{yield}{cost},$$

where $yield = Y(x_1, x_2, \ldots, x_m) = \frac{\sum_{i=1}^{m} y(x_i)}{m}$,

$$y(x_i) = \begin{cases} 1 & \text{if } x_i \text{ satisfies its SLA;} \\ e^{-p*(|\frac{x_i - x_i^{'}}{x_i^{'}}|)} - 1 & \text{otherwise,} \end{cases}$$

and $cost = 1 + \frac{\sum_{i=1}^{n}(1-u_i^k)^{\frac{1}{k}}}{n}$. Note that the metric *yield* is a summarized gain over m performance metrics $x_1, x_2, \cdots, x_m$. The utility function $y(x_i)$ decays when metric

x_i violates its performance objective $x_i^{'}$ in SLA. *cost* is calculated as the summarized utility based on n utilization status $u_1, u_2, \cdots, u_n$. Both the utility functions decay under the control of the decay factors of p and k, respectively. We consider throughput and response time as the performance metrics and u_{cpu}, u_{io}, u_{mem}, u_{swap} as the utilization metrics. The *reward* punishes any capacity setting that violates the SLA and gives incentives to high resource efficiency.

Self-adaptive Learning Engine

At the heart of iBalloon is a self-adaptive learning agent responsible for each VM's capacity management. Reinforcement learning is concerned with how an agent ought to take actions in a dynamic environment so as to maximize a long term reward [67]. It fits naturally within iBalloon's feedback driven, interactive framework. RL offers opportunities for highly autonomous and adaptive capacity management in cloud dynamics. It assumes no priori knowledge about the VM's running environment. It is able to capture the delayed effect of reconfigurations to a large extent.

A RL problem is usually modeled as a *Markov Decision Process* (MDP). Formally, for a set of environment states $\mathcal{S}$ and a set of actions $\mathcal{A}$, the MDP is defined by the transition probability $P_a(s, s') = Pr(s_{t+1} = s' | s_t = s, a_t = a)$ and an immediate reward function $R = E[r_{t+1} | s_t = s, a_t = a, s_{t+1} = s']$. At each step t, the agent perceives its current state $s_t \in \mathcal{S}$ and the available action set $\mathcal{A}(s_t)$. By taking action $a_t \in \mathcal{A}(s_t)$, the agent transits to the next state s_{t+1} and receives an immediate reward r_{t+1} from the environment. The value function of taking action a in state s can be defined as:

$$Q(s, a) = E\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\},$$

where $0 \leq \gamma < 1$ is a discount factor helping $Q(s, a)$'s convergence. The optimal

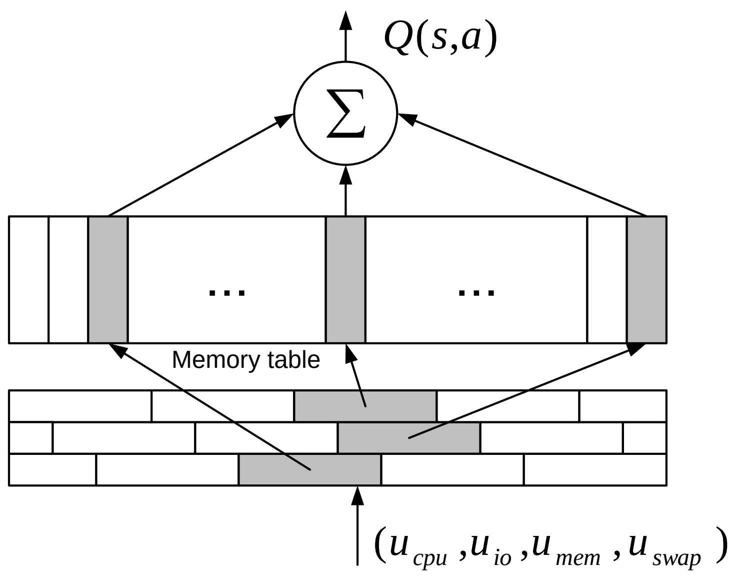


Figure 6.3: CMAC-based Q table.

policy is as simple as: always select the action a that maximizes the value function $Q(s,a)$ at state s. Finding the optimal policy is equivalent to obtain an estimation of $Q(s,a)$ which approximates its actual value. The estimate of $Q(s,a)$ can be updated each time an interaction (s_t, a_t, r_{t+1}) is finished:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha * [r_{t+1} + \gamma * Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)],$$

where α is the learning rate. The interactions consist of exploitations and explorations. Exploitation is to follow the policy obtained so far; in contrast, exploration is the selection of random actions to capture the change of environment so as to refine the existing policy. We follow the ϵ-greedy policy to design the RL agent. With a small probability ϵ, the agent picks up a random action, and follows the best policy it has found for the rest of the time.

In VM capacity management, the state s corresponds to the VM's running status and action a is the management operation. For example, the action a can show in the form of $(nop, increase, decrease)$, which indicates an increase in the VM's memory size and a decrease in I/O bandwidth. Actions in continuous space remains an open research problem in the RL field, we limit the RL agent to discrete actions. The actions are discretized by setting steps on each resource instead. VCPU is incremented or decremented by one at a time and memory is reconfigured in a step of 256MB. I/O bandwidth is changed by a step of 0.5MB.

The requirement of autonomy in VM capacity management poses two key questions on the design of the RL engine. First, how to overcome the scalability and adaptability problems in RL? Second, how would the multiple RL agents, each of which represents a VM, coordinate and optimize system-wide performance? We answer the questions by designing the VM capacity management agent as a distributed RL agent with a highly efficient representation of the Q table. Unlike, multi-agent RL, in which each agent needs to maintain other competing agents' information, distributed RL does not have explicit coordination scheme. Instead, it relies on the feedback signals for coordination. For example, when resources are contented, negative feedbacks help resolve the contention. VMs belonging to the same application receive the same feedback, which coordinates resource allocations in the virtual cluster. An immediate benefit of distributed learning is that the complexity of the learning problem does not grow exponentially with the number of VMs.

The VM running status is naturally defined in a multi-dimensional continuous space. Although we limit the actions to be discrete operations, the state itself can render the Q value function intractable. Due to its critical impact on the learning performance, there are many studies on the Q function representation [67, 68]. We carefully reviewed these works and decided to borrow the design in the Cerebellar Model Articulation Controller (CMAC) [3] to represent the Q function. It maintains multiple coarse-grained Q tables or so-called tiles, each of which is shifted by a random offset with respect to each other. With CMAC, we can achieve higher resolution in the Q table with less cost. For example, if each status input (an element in the status vector) is discretized to five intervals (a resolution of 20%), 32 tiles will give a resolution less than 1% (20%/32). The total size of the Q tables is reduced to $32 * 5^4$ compared to the size of 100^4 if plain look-up table is used. In CMAC, the actual Q table is stored in a one-dimensional memory table and each cell in the table stores a weight value. Figure Figure 6.3 illustrates the architecture of a one-dimensional CMAC. The VM running status listed in Figure Figure 6.3 is only for illustration purpose. The state needs to work with a four-dimensional CMAC. Given a state s, CMAC uses a hash function, which takes a pair of state and action as input, to generate indexes for the (s, a) pair. CMAC uses the indexes to access the memory cells and calculates $Q(s, a)$ as the sum of the weights in these cells.

One advantage of CMAC is its efficiency in handling limited data. Similar VM states will generate CMAC indexes with a large overlap. Thus, updates to one state can generalize to the others, leading to accelerated RL learning process. One update of the CMAC-based Q table only needs 6.5 milliseconds in our testbed, in comparison with the 50-second update time in a multi-layer neural network [57]. Once a VM finishes an iteration, it submits the four-tuple (s_t, a_t, s_{t+1}, r_t) to `Decision-maker`. Then the corresponding RL agent updates the VM's Q table using Algorithm 3. In the algorithm, we further enhanced the CMAC-based Q table with fast adaptation when SLA violated. We set the learning rate α to 1 whenever receives a negative penalty. This ensures that "bad" news travels faster than good news allowing the learning agent quickly response to the performance violation.

Algorithm 3 Update the CMAC-based Q value function

```
1: **Input** $s_t$, $a_t$, $s_{t+1}$, $r_t$;
2: **Initialize** $\delta = 0$;
3: $I[a_t][0] = get\_index(s_t)$;
4: $Q(s_t, a_t) = \sum_{j=1}^{j \leq num\_tilings} Q[I[a_t][j]]$;
5: $a_{t+1} = get\_next\_action(s_{t+1})$;
6: $I[a_{t+1}][0] = get\_index(s_{t+1})$;
7: $Q(s_{t+1}, a_{t+1}) = \sum_{j=1}^{j \leq num\_tilings} Q[I[a_{t+1}][j]]$;
8: $\delta = r_t - Q(s_t, a_t + \gamma * Q(s_{t+1}, a_{t+1}))$;
9: for $i = 0; i < num\_tilings; i++$ do
10:    /*If SLA violated, enable fast adaptation*/
11:    if $r_t < 0$ then
12:       $\theta[I[a_t][i]] += (1.0/num\_tilings) * \delta$;
13:    else
14:       $\theta[I[a_t][i]] += (\alpha/num\_tilings) * \delta$;
15:    end if
16: end for
```

6.4 Implementation

iBalloon has been implemented as a set of user-level daemons in guest and host OSes. The communication between the host and guest VMs is carried out through an inter-domain channel. In our Xen-based testbed, we used `Xenstore` for the host and guest information exchange. Xenstore is a centralized configuration database that is accessible by all domains on the same host. The domains who are involved in the communication place "watches" on a group of pre-defined keys in the database. Whenever sender initializes a communication by writing to the key, the receiver is notified and possibly trigging a callback function. Upon a new VM joining a host, `Host-agent`, one per machine, creates a new key under the VM's path in Xenstore. `Host-agent` launches a worker thread for the VM and the worker "watches" any change of the key. Whenever a VM submits a resource request via the key, the worker thread retrieves the request details and activates the corresponding handler in `dom0` to handle the request. The VM receives the feedback from `Host-agent` in a similar way.

We implemented resource allocation in `dom0` in a synchronous way. VMs send out resource requests in a fixed interval (30 second in our experiments) and `Host-agent` waits for all the VMs before satisfying any request. It is often desirable to allow users to submit requests with different management intervals for flexibility and reliability in resource allocation. We leave the extension of iBalloon to asynchronous resource allocation in the future work. After VMs and `Host-agent` agree on the resource allocations, `Host-agent` modifies individual VMs' configurations accordingly. We changed the memory size of the VM by writing the new size to the domain's `memory/target` key in Xenstore. VCPU number was altered by turning on/off individual CPUs via key `cpu/CPUID/availability`. For I/O bandwidth control, we used command `lsof` to correlate VMs' virtual disks to processes and change the corresponding processes' bandwidth allocation via the Linux device-mapper driver `dm-ioband` [70].

`App-agent`, one per host, maintains the hosted application SLA profiles. In our experiments, it periodically queries participant machines through standard socket communication and reports application performance, such as throughput and response time, to `Host-agent`. In a more practical scenario, the application performance should be reported by a third-party application monitoring tool instead of the clients. iBalloon can be easily modified to integrate such tools. We implemented the `Decision-maker` as a process residing in each guest OS. The learning process is local to individual VMs and incurs computation and storage overhead. The distributed implementation of `Decision-maker` ensures that the scalability of iBalloon is not limited by the number of VMs. Quantitative comparison of the distributed implementation and a centralized approach will be presented in Section 6.6.5.

We use `xentop` utility to report VM CPU utilization. `xentop` is instrumented to redirect the utilization of each VM to separate log files in the `tmpfs` folder `/dev/shm` every second. A small utility program parses the logs and calculates the average CPU utilization for every management interval. The disk I/O utilization is calculated as a ratio of achieved bandwidth to allocated bandwidth. The achieved the bandwidth can be obtained by monitoring the disk activities in `/proc/PID/io`. PID is the process number of a VM's virtual disk in `dom0`. The swap rate can also be collected in a similar way. We consider memory utilization to be the guest OS metric `Active` over memory size. The `Active` metric in `/proc/meminfo` is a coarse estimate of actively used memory size. However, it is lazily updated by guest kernel especially during memory idle periods. We combine the guest reported metric and swap rate for a better estimate of memory usage. With explorations from the learning engine, iBalloon has a better chance to reclaim idle memory without causing significant swapping.

6.5 Experiment Design

6.5.1 Methodology

To evaluate the efficacy of iBalloon, we attempt to answer the following questions: (1) How well does iBalloon perform in the case of single VM capacity management? Can the learned policy be re-used to control a similar application or on a different platform? (Section 6.6.3) (2) When there is resource contention, can iBalloon properly distribute the constrained resource and optimize overall system performance? (Section 6.6.4) (3) How is iBalloon's scalability and overhead? (Section 6.6.5) We selected three representative server workloads as the hosted applications. TPC-W [72] is an E-Commerce benchmark that models after an online book store, which is primary CPU-intensive. It consists of two tiers, i.e. the front-end application (APP) tier and the back-end database (DB) tier. SPECweb [71] is a web server benchmark suite that delivers dynamic web contents. It is a CPU and network-intensive server application. TPC-C [72] is an online transaction processing benchmark that contains lightweight disk reads and sporadic heavy writes. Its performance is sensitive to memory size and I/O bandwidth.

To create dynamic variations in resource demand, we instrumented the workload generators of TPC-W and TPC-C to change client traffic level at run-time. The workload generator reads the traffic level from a trace file, which models after the real Internet traffic pattern [69]. We scaled down the Internet traces to match the capacity of our platform.

6.5.2 Testbed Configurations

Two clusters of nodes were used for the experiments. The first cluster (CIC100) is a shared research environment, which consists of a total number of 22 DELL and SUN machines. Each machine in CIC100 is equipped with 8 CPU cores and 8GB memory. The CPU and memory configurations limit the number of VMs that can be consolidated on each machine. Thus, we use CIC100 as a resource constrained cloud testbed to verify iBalloon's effectiveness for small scale capacity management. Once iBalloon gains enough experiences to make decisions, we applied the learned policies to manage a large number of VMs. CIC200 is a cluster of 16 DELL machines dedicated to the cloud management project. Each node features a configuration of 12 CPU cores (with hyperthreading enabled) and 32 GB memory. In the scale-out testing, we deployed 64 TPC-W instances, i.e. a total number of 128 VMs on CIC200. To generate sufficient client traffic to these VMs, all the nodes in CIC100 were used to run client generators, with 3 copies running on each node.

We used Xen version 4.0 as our virtualization environment. `dom0` and guest VMs were running Linux kernel 2.6.32 and 2.6.18, respectively. To enable on-the-fly reconfiguration of CPU and memory, all the VMs were para-virtualized. The VM disk images were stored locally on a second hard drive on each host. We created the `dm-ioband` device mapper on the partition containing the images to control the disk bandwidth. For the benchmark applications, MySQL, Tomcat and Apache were used for database, application and web servers.

6.6 Experimental Results

6.6.1 Evaluation of the Reward Metric

The *reward* metric synthesizes multi-dimensional application performance and resource utilizations. We are interested in how the *reward* signal guides the capacity management. The decay rates p and k reflect how important it is for an application to meet the performance objectives in its SLA and how aggressive the user increase resource utilization even at the risk of overload.

We decided to guarantee user satisfaction and assume risk neutral users, and set $p = 10$ and $k = 1$. Figure Figure 6.5 shows how the *reward* reflect the status of VM capacity. In this experiment, we varied the client traffic to occasionally exceed the VM's capacity. *reward* is calculated from the DB tier of a TPC-W instance, with a fixed configuration of 3 VCPU, 2GB memory and 2 MB/s disk bandwidth. As shown in Figure Figure 6.5, when the load is light, performance objectives are met. During this period, *yield* is set to 1 and *cost* dominates the change of *reward*. As traffic increases, resource utilization goes up incurring smaller *cost*. Similarly, *reward* drops when traffic goes down because of the increase of the *cost* factor. In contrast, when the VM becomes overloaded with SLA violations, the factor of *yield* dominates *reward*

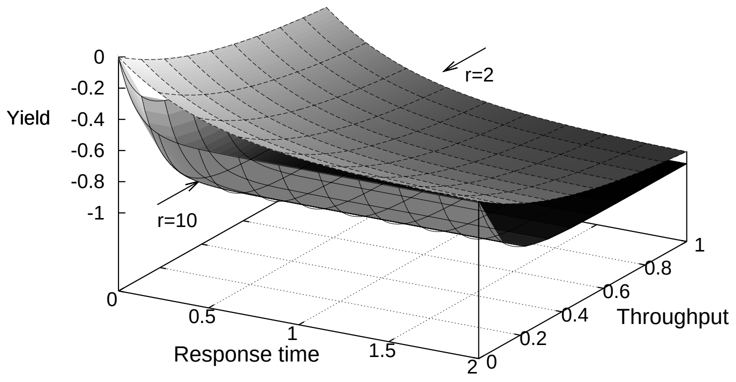


Figure 6.4: Application yield with different decay rates.

by imposing a large penalty. In conclusion, *reward* effectively punishes performance violations and gives users incentives to release unused resources.

6.6.2 Exploitations vs. Explorations

Reinforcement learning is a direct adaptive optimal control approach which relies on the interactions with the environment. Therefore, the performance of the learning algorithm depends critically on how the interactions are defined. Explorations are often considered as sub-optimal actions that lead to degraded performance. However, without enough explorations, the RL agent tends to be trapped in local optimal policies, failing to adapt to the change of the environment. On the other hand, too much exploration would certainly result in unacceptable application performance. Before iBalloon is actually deployed, we need to determine the value of exploration rate, that best fits our platform.

In this experiment, we dedicated a physical host to one application and initialized the VM's Q table to all zeros. We varied the exploration rate of the learning algorithm and draw the application performance of TPC-W in Figure Figure 6.6. The bars represent the average of 5 one-hour runs with the same exploration rate and variations.

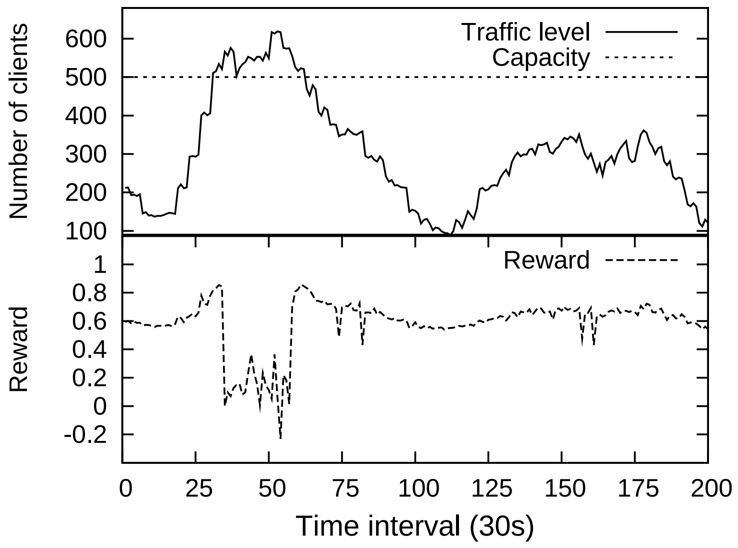


Figure 6.5: Reward with different traffic levels.

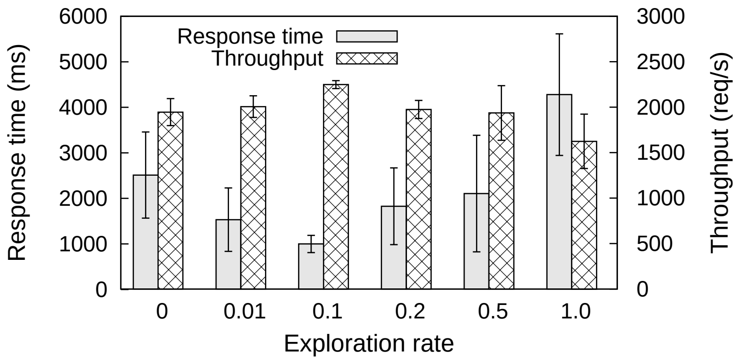
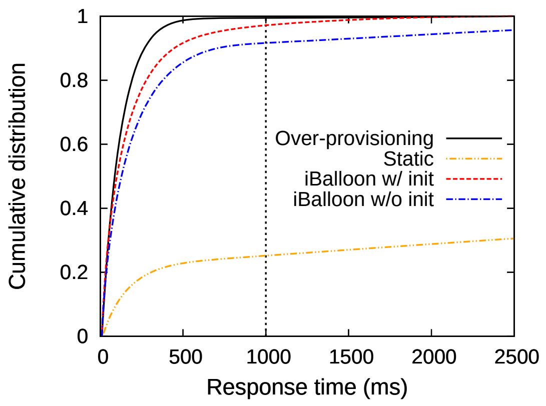
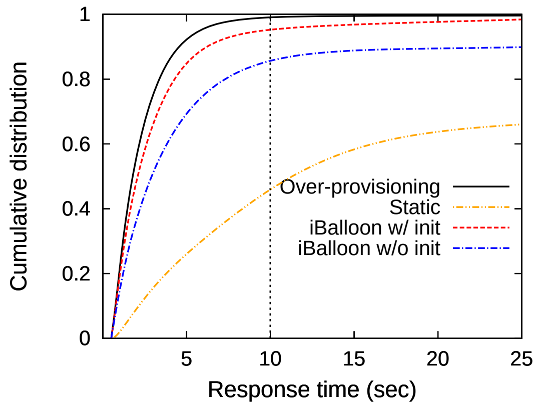


Figure 6.6: Application performance with different exploration rates.

From the figure, we can see that the response time of TPC-W is convex with respect to the exploration rate with $\epsilon = 0.1$ being the optimal. The same exploration rate also gives the best throughput as well as the smallest variations. Experiments with TPC-C suggested a similar exploration rate. We also empirically determined the learning rate and discount factor. For the rest of this chapter, we set the RL parameters to the following values: $\epsilon = 0.1, \alpha = 0.1, \gamma = 0.9$.



(a) TPC-W



(b) TPC-C

Figure 6.7: Response time under various reconfiguration strategies.

6.6.3 Single Application Capacity Management

In its simplest form, iBalloon manages a single VM or application's capacity. In this subsection, we study its effectiveness in managing different types of applications with distinct resource demands. The RL-based auto-configuration can suffer from initial poor performance due to explorations with the environment. To have a better understanding of the performance of RL-based capacity management, we tested two variations of iBalloon, one with an initialization of the management policy and one without. We denote them as *iBalloon w/ init* and *iBalloon w/o init*, respectively. The initial policy was obtained by running the application workload for 10 hours, during which iBalloon interacted with the environment with only exploration actions.

Figure Figure 6.7(a) and Figure Figure 6.7(b) plot the performance of iBalloon and its variations in a 5-hour run of the TPC-W and TPC-C workloads. Note that during each experiment, the host was dedicated to TPC-W or TPC-C, thus no resource contention existed. In this simple setting, we can obtain the upper bound and lower bound of iBalloon's performance. The upper bound is due to resource *over-provisioning*, which allocates more than enough resource for the applications. The lower bound performance was derived from a VM template whose capacity is not changed during the test. We refer it as *static*. We configured the VM template with 1 VCPU and 512 MB memory in the experiment. If not otherwise specified, we used the same template for all VM default configuration in the remaining of this chapter.

From Figure Figure 6.7(a), we can see that, iBalloon achieved close performance compared with *over-provisioning*. *iBalloon w/o init* managed to keep almost 90% of the request below the SLA response time threshold except that a few percent of requests had wild response times. It suggests that, although started with poor policies, iBalloon was able to quickly adapt to good policies and maintained the performance at a stable level. We attribute the good performance to the highly efficient representation of the Q table. The CMAC-enhanced Q table was able to generalize to the continuous state space with a limited number of interactions. Not surprisingly, *static*'s poor result again calls for appropriate VM capacity management.

As shown in Figure Figure 6.7(b), *iBalloon w/ init* showed almost optimal performance for TPC-C workload too. But without policy initialization, iBalloon can only prevent around 80% of the requests from SLA violations; more than 15% requests would have response times larger than 30 seconds. This barely acceptable performance stresses the importance of a good policy in more complicated environments. Unlike CPU, memory sometimes shows unpredictable impact on performance. The dead time due to the factor of memory is much longer than CPU (10 minutes compared to 5 minutes in our experiments). In this case, iBalloon needs a longer time to obtain a good policy. Fortunately, the derived policy, which is embedded in the Q table, can be possibly re-used to manage similar applications.

Table Table 6.2 lists the application improvement if the learned management policies are applied to a different application or to a different platform. The improvement is calculated against the performance of iBalloon without an initial policy.

Table 6.2: Performance improvement due to initial policy learned from different applications and cloud platforms.

	Throughput	Response time
Trained in TPC-W Tested in SPECweb	40%	80%
Trained in CIC100 Tested in CIC200	20%	30%

SPECweb [71] is a web server benchmark suite that contains representative web workloads. The E-Commerce workload in SPECweb is similar to TPC-W (CPU-intensive) except that its performance is also sensitive to memory size. Results in Table Table 6.2 suggest that the Q-table learned for TPC-W also worked for SPECweb. An examination of iBalloon's log revealed that the learned policy was able to successfully match CPU allocation to incoming traffic. A policy learned on cluster CIC100 can also give more than 20% performance improvement to the same TPC-W application on cluster CIC200. Given the fact that the nodes in CIC100 and CIC200 have more than 30% difference on CPU speed and disk bandwidth, we conclude that iBalloon policies are applicable to heterogeneous platforms across cloud systems.

The *reward* signal provides strong incentives to give up unnecessary resources. In Figure Figure 6.8, we plot the configuration of VCPU, memory and I/O bandwidth of TPC-W, SPECweb and TPC-C as client workload varied. Recall that we do not have an accurate estimation of memory utilization. We rely on the `Active` metric in `meminfo` and the swap rate to infer memory idleness. The Apache web server used in SPECweb periodically free unused `httpd` process thus memory usage information in `meminfo` is more accurate. As shown in Figure Figure 6.8, with a 10-hour trained policy, iBalloon was able to expand and shrink CPU and I/O bandwidth resources as workload varied. As for memory, iBalloon was able to quickly respond to memory

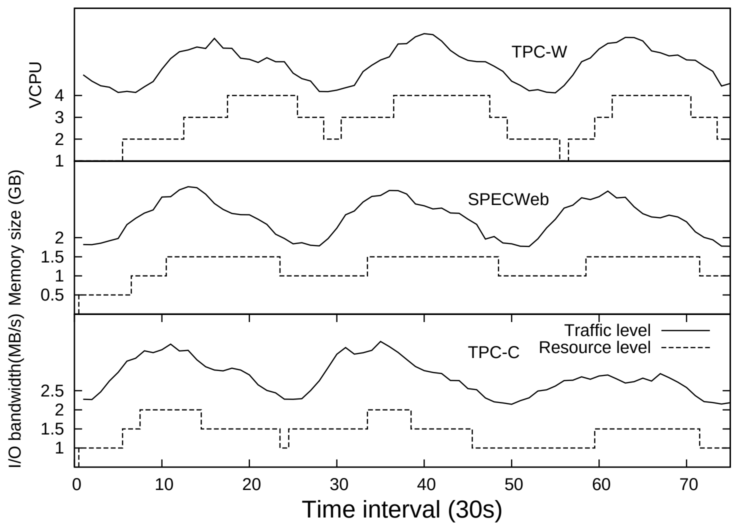
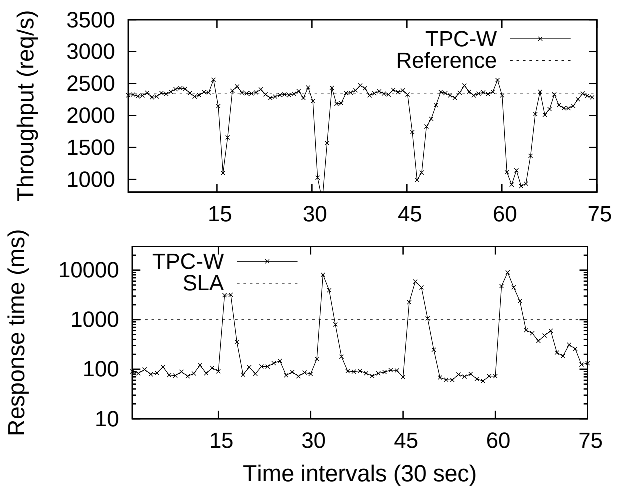


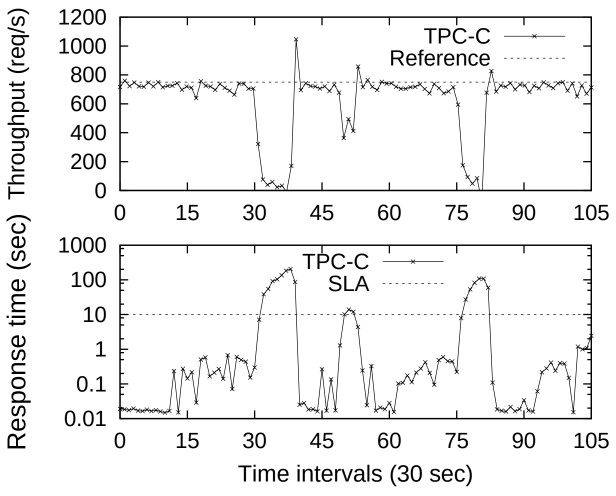
Figure 6.8: Resources allocations changing with workload.

pressure; it can release part of the unused memory although not completely. The agreement in shapes of each resource verifies the accuracy of the *reward* metric.

We note that the above results only show the performance of iBalloon statistically. In practice, service providers concern more about user-perceived performance, because in production systems, mistakes made by autonomous capacity management can be prohibitively expensive. To test iBalloon's ability of determining the appropriate capacity online, we ran the workload generators at full speed and reduced the VM's capacity every 15 management intervals. Figure 6.6.3 plots the client-perceived results in TPC-W and TPC-C. In both experiments, iBalloon was configured with initial policies. Each point in the figures represents the average of a 30-second management interval. As shown in Figure Figure 6.9(a), iBalloon is able to promptly detect the mis-configurations and reconfigure the VM to appropriate capacity. On average, throughput and response time can be recovered within 7 management intervals. Similar results can also be observed in Figure Figure 6.9(b) except that



(a) TPC-W



(b) TPC-C

Figure 6.9: User-perceived performance under iBalloon.

client-perceived response times have larger fluctuations in TPC-C workload.

6.6.4 Coordination in Multiple Applications

iBalloon is designed as a distributed management framework that handles multiple applications simultaneously. The VMs rely on the feedback signals to form their capacity management policy. Different from the case of a single application, in which the feedback signal only depends on the resource allocated to the hosting VM, in multiple application hosting, the feedback signals also reflect possible performance interferences between VMs.

We designed experiments to study iBalloon's performance in coordinating multiple applications. Same as above, iBalloon was configured to manage only the DB tiers of TPC-W workload. All the DB VMs were homogeneously hosted in one physical host while the APP VMs were over-provisioned on another node. The baseline VM capacity strategy is to statically assign 4VCPU and 1GB memory to all the DB VMs, which is considered to be over-provisioning for one VM. iBalloon starts with a VM template, which has 1VCPU and 512MB memory. Figure Figure 6.10 draws the performance of iBalloon normalized to the baseline capacity strategy in a 5-hour test. The workload to each TPC-W instances varied dynamically, but the aggregated resource demand is beyond the capacity of the machine that hosts the DB VMs. Figure Figure 6.10 shows that, as the number of the DB VMs increases, iBalloon gradually beats the baseline in both throughput and response time.

An examination of the iBalloon logs revealed that iBalloon suggested a smaller number of VCPUs for the DB VMs, which possibly alleviated the contention for CPU. The baseline strategy encouraged resource contention and resulted in wasted work. In summary, iBalloon, driven by the feedback, successfully coordinated competing VMs to use the resource more efficiently.

6.6.5 Scalability and Overhead Analysis

We scaled iBalloon out to the large dedicated CIC200 cluster and deployed 64 TPC-W instances, each with two tiers, on the cluster. We randomly deployed the 128 VM on the 16 nodes, assuming no topology information. To avoid possible hotspot and load unbalancing, each node hosted 8 VMs, 4 APP and 4 DB tiers. We implemented `Decision-maker` as distributed decision agents. The deployment is challenging to autonomous capacity management for two reasons. First, iBalloon ought to coordinate VMs on different hosts, each of which runs its own resource allocation policy. The dependent relationships makes it harder to orchestrate all the VMs. Second, consolidating APP (network-intensive) tiers with DB (CPU-intensive) tiers onto the same host poses challenges in finding the balanced configuration.

Figure Figure 6.11 plots the average performance of 64 TPC-W instances for a 10-hour test. In addition to iBalloon, we also experimented with four other strategies.

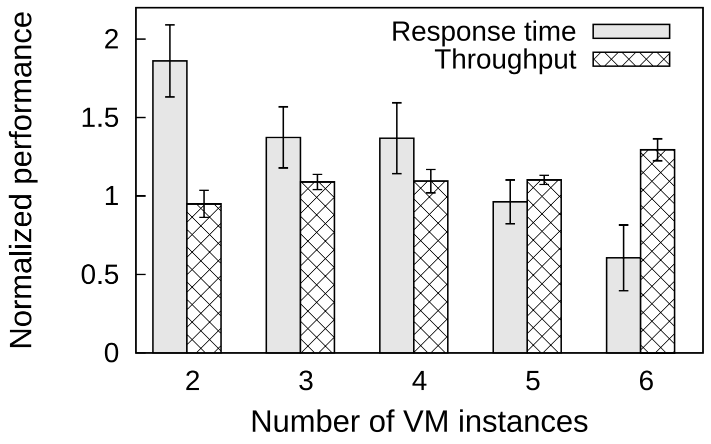


Figure 6.10: Performance of multiple applications due to iBalloon.

The *optimal* strategy was obtained by tweaking the cluster manually. It turned out that the setting: DB VM with 3VCPU,1GB memory and APP VM with 1VCPU, 1GB memory delivered the best performance. *work-conserving* scheme is similar to the baseline in last subsection; it sets all VMs with fixed 4VCPU and 1GB memory. *Adaptive proportional integral (PI)* method [52] directly tracks the error of the measured response time and the SLO and adjusts resource allocations to minimize the error. *Auto-regressive-moving-average (ARMA)* method [51] builds a local linear relationship between allocated resources and response time with recently collected samples, from which the resource reconfiguration is calculated. The performance is normalized to *optimal*. For throughput, higher is better; for response time, lower is better.

From the figure, iBalloon achieved close throughput as *optimal* while incurred 20% degradation on request latency. This is understandable because any change in a VM's capacity, especially memory reconfigurations, bring in unstable periods. iBalloon outperformed *work-conserving* scheme by more than 20% in throughput. Although *work-conserving* had compromised throughput, it achieved similar response time as

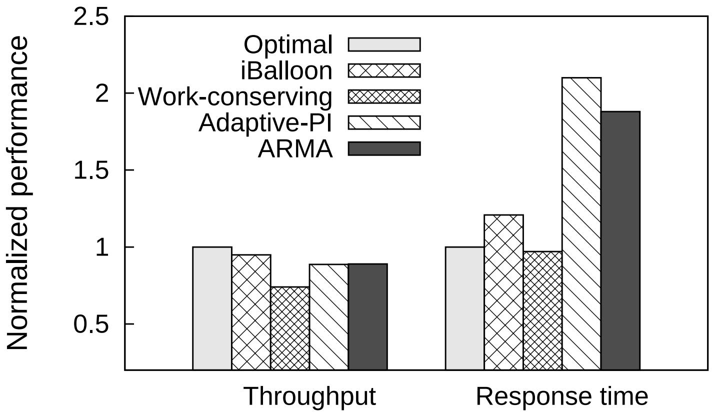


Figure 6.11: Performance due to various reconfiguration approaches on a cluster of 128 correlated VMs.

optimal because it did not perform any reconfigurations. *adaptive-PI* and *ARMA* achieved similar throughput as iBalloon but with more than 100% degradations on response time. These control methods which are based either on system identification or local linearization can suffer poor performance under both workload and cloud dynamics. We conclude that iBalloon scales to 128 VMs on a correlated cluster with near-optimal application performance. In the next, we perform tests to narrow down the overhead incurred on request latency.

In previous experiments, iBalloon incurred non-negligible cost in response time. The cost was due to the real overhead of iBalloon as well as the performance degradation caused by the reconfiguration. To study the overhead incurred by iBalloon, we repeated the experiment as in Section Figure 6.11 except that iBalloon operated on the VMs with optimal configurations and reconfigurations were disabled in `Host-agent`. In this setting, the overhead only comes from the interactions between VMs and iBalloon. Figure Figure 6.12 shows the overhead of iBalloon with two different implementations of `Decision-maker`, namely the centralized and the distributed implementations. In the centralized approach, a designated server performs RL algo-

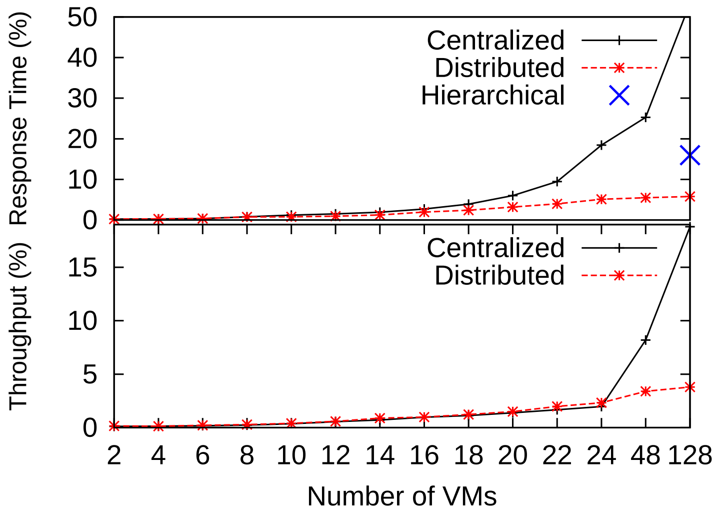


Figure 6.12: Runtime overhead of iBalloon.

rithms for all the VMs. Again, the overhead is normalized to the performance in the *optimal* scheme.

Figure Figure 6.12 suggests that the centralized decision server becomes the bottleneck with as much as 50% overhead on request latency and 20% on throughput as the number of VMs increases. In contrast, the distributed approach, which computes capacity decisions on local VMs, incurred less than 5% overhead on both response time and throughput. To further confirm the limiting factor of centralized decision server, we split the centralized decision work onto two separate machines(denoted as Hierarchical) in the case of 128 VMs. As shown in Figure Figure 6.12, the overhead on request latency reduces by more than a half. Additional experiments revealed that computing the capacity management decisions locally in VMs requires no more than 3% CPU resources for Q computation and approximately 18MB of memory for Q table storage. The resource overhead is insignificant compared to the capacity of the VM template (1VCPU, 512MB). These results conclude that iBalloon adds no more than 5% overhead to the application performance with a manageable resource cost.

6.7 Summary

In this work, we present *iBalloon*, a generic framework that allows self-adaptive virtual machine resource provisioning. The heart of iBalloon is the distributed reinforcement learning agents that coordinate in dynamic environment. Our prototype implementation of iBalloon, which uses a highly efficient reinforcement learning algorithm as the learning, was able to find the near optimal configurations for a total number of 128 VMs on a closely correlated cluster with no more than 5% overhead on application throughput and response time.

Chapter 7

Conclusions and Future Work

This dissertation aims to build an automatic cloud resource management system. In this chapter, we summarize the approaches presented in this dissertation and give the directions for future work.

7.1 Conclusions

Although cloud computing has gained sufficient popularity recently, there are still some key impediments to large scale enterprise adoption. Cloud management is one of the top challenges. We consider the resource management problem in cloud computing as a capacity management problem for individual hosted applications.

Understanding server capacity is crucial to system capacity planning and QoS-aware resource management. Different from traditional web hosting, capacity planning for applications in a cloud should take place on a continuous basis during the life time of the application. To guarantee application SLA and achieve efficient resource usage, cloud capacity management requires accurate and precise capacity measurement. We study the measurement of capacity for a complex scenario, multi-tier websites with dynamic workloads. Traditional application-level metrics are limited in measurement accuracy and timeliness. We propose a derived PI metrics based on low-level statistics to monitor the system progress in order to determine when more capacity is needed. We build a coordinated capacity predictor based on snapshots taken from low-level metrics. Results on a multi-tier E-commerce benchmark show accurate and responsive measurement of system capacity.

Although offering cloud users the illusion of an infinite on-demand resource pool, cloud providers do not guarantee consistent performance over time and to users in different regions. Considerable performance variations have been observed from leading cloud providers due to background resource scheduling and multiplexing. This poses challenges on automated resource management linking application performance to resource configurations. We demonstrate that under cloud dynamics, it is difficult to determine a static relationship between resource and performance. To address this problem, we propose a model-independent fuzzy control based approach for CPU allocation. For adaptive and stable control performance, we embed the controller with self-tuning output amplification and flexible rule selection. Finally, we build a QoS provisioning framework that supports multi-objective QoS control and service differentiation. Experiments on a virtual cluster with two service classes show the effectiveness of our approach in performance and power control.

Capacity management in cloud involves the management of multiple virtualized resources. Model-free feedback control approaches can not be easily extended to multi-dimensional resource allocation due to complex interplay between resources. The control approach can also be affected by process delays in resource reconfiguration. We consider the capacity management as a decision-making problem and employ reinforcement learning to optimize the process. The optimization depends on the trial-and-error interactions with the cloud system. In order to improve the initial management performance, we propose a model-based RL algorithm. The neural network based environment model, which is learned from previous management history, generates simulated resource allocations for the RL agent. Experiment results on heterogeneous applications show that our approach makes efficient use of limited interactions and find near optimal resource configurations within 7 steps.

The existence of virtual cluster applications and their arbitrary and changing deployment on the physical nodes poses challenges in capacity management. Centralized approach can not scale as the number of VMs increases. We decompose the capacity management of hosted applications into sub-problems concerning individual VMs. We propose a distributed learning mechanism which treats each VM as a highly autonomous agent. The heart of the approach is a RL algorithm with efficient representation of experiences. We design the RL state space on VM running status to accommodate workload dynamics and define the reward signal based on the PI metric. We prototype the mechanism and test with a real system. Our approach show good scalability and performance on a closely correlated cluster with 128 VMs.

7.2 Future Directions

There are several issues and new directions along the line of this dissertation. In this dissertation work, we consider the resource management problem as a vertical resource allocation problem, in which resources are added or removed in a single VM. Another way to manage the capacity of hosted applications is horizontal scaling. For certain type of applications such as MapReduce and high performance applications, it may be beneficial to use horizontal scaling due to the presence of application-level task scheduling. This type of task scheduling accommodates node join and leave, and work especially well on cluster of homogeneous nodes. Horizontal scaling allocates resources in a relatively coarse granularity, usually in terms of virtual machines. The lead time in horizontal scaling is several minutes compared to sub-second in vertical resource allocation. There are interesting problems need investigation in horizontal scaling. First, how to design policies for horizontal resource allocation that minimize application-level performance degradation considering the high cost in allocation. Second, leading cloud providers like Amazon EC2 charge cloud usage on a hourly and per VM basis. How to balance the trade-off between performance and rental cost for a specific application deserves further study.

There is another cloud resource management operation we did not consider in this dissertation, that is VM migration. Although in Chapter 6, we decompose the resource allocation problem into a per VM level which facilitates the integration of migration operations into the management. There are still some important issues and challenges specific to VM migration. First, migrations come with cost, usually in terms of application downtime. The cost associate with a migration is dependent on the characteristics of the migrated VM and the application within it. The first interesting problem pertain to the selection of migration candidate that minimizes the application downtime and possibly the impact to other co-running VMs. Second, the selection of the migration destination affects the resource management in the cluster. Migrating to lightly loaded nodes improves load balancing; migrating to a fewer number of busy nodes can possibly reduce the energy consumption by turning off unused nodes. Finally, as the number of physical nodes increases, the optimal selection of the migration target will cause significant overhead and is sometimes impossible. How to make reasonably good migration decisions with only local information warrants future research effort.

We consider the cloud as a black-box and design application-oriented resource management mechanism to offset the cloud dynamics. The study of cloud SLAs itself introduces a broad range of interesting problems. Leading cloud providers like Amazon EC2 only includes availability objectives in its cloud SLA. No performance guarantees are specified. Due to background hardware scheduling and multiplexing, it is difficult to guarantee consistent experiences to cloud users, especially when the cloud infrastructure contains heterogeneous hardware. Even within a single machine, it is still non-trivial to provide consistent performance. With the advances in CPU architecture, such as heterogeneous cores, non-uniform memory access (NUMA) and simultaneous hardware threading (SMT), there are many more sources of performance variation. Two possible directions are: (1) architecture and user-aware scheduling of virtualized resources that guarantees fairness among users. (2) By accepting the performance variation due to background scheduling, we perform better accounting of resource usage and provide cost-proportional computing to cloud users.

REFERENCES

[1] http://www.research.ibm.com/autonomic.

[2] T. F. Abdelzaher, K. G. Shin, and N. Bhatti. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Trans. Parallel Distrib. Syst.*, 13, January 2002.

[3] J. S. Albus. A New Approach to Manipulator Control: the Cerebellar Model Articulation Controller (CMAC). *Journal of Dynamic Systems, Measurement, and Control*, 1975.

[4] Amazon Spot Instances. http://aws.amazon.com/ec2/spot-instances.

[5] C. Amza, E. Cecchet, A. Chanda, A. L. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and implementation of dynamic web site benchmarks. In *Proc. of WWC*, 2002.

[6] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical report, EECS Department, University of California, Berkeley, Feb 2009.

[7] C. G. Atkeson and J. C. Santamar'ia. A comparison of direct and model-based reinforcement learning. In *In International Conference on Robotics and Automation*, 1997.

[8] G. Banga and P. Druschel. Measuring the capacity of a web server. In *Proc. of USITS*, 1997.

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, 2003.

[10] X. Bu, J. Rao, and C.-Z. Xu. A reinforcement learning approach to online web systems auto-configuration. In *ICDCS*, 2009.

[11] G. Candea, E. Kiciman, S. Kawamoto, and A. Fox. Autonomous recovery in componentized internet applications. *Cluster Computing*, 2006.

[12] J. P. Cassaza, M. Greenfield, and K. shi. Redefining server performance characterization for virtualization benchmarking. In *Intel technology Journal*, 2006.

[13] C.-L. Chen. Ieee 802.11e edca qos provisioning with dynamic fuzzy control and cross-layer interface. In *ICCCN*, 2007.

[14] H. Chen, G. Jiang, H. Zhang, and K. Yoshihira. Boosting the performance of computing systems through adaptive configuration tuning. In *SAC*, 2009.

[15] H. Chen and P. Mohapatra. Session-based overload control in qos-aware web servers. In *Proc. of INFOCOM*, 2002.

[16] X. Chen, P. Mohapatra, and H. Chen. An admission control scheme for predictable server response time for web accesses. In *Proc. of WWW*, 2001.

[17] L. Cherkasova and P. Phaal. Session based admission control: a mechanism for improving the performance of an overloaded web server. Technical Report HPL-98-119, HP Labs, 1998.

[18] L. Cherkasova and L. Staley. Measuring the capacity of a streaming media server in a utility data center environment. In *Proc. of ACM Multimedia*, 2002.

[19] I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proc. of OSDI*, 2004.

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

[21] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *SOSP*, 2005.

[22] J. Dejun, G. Pierre, and C.-H. Chi. EC2 performance analysis for resource provisioning of service-oriented applications. In *Proceedings of the 3rd Workshop on Non-Functional Properties and SLA Management in Service-Oriented Computing*, 2009.

[23] Y. Diao, N. Gandhi, J. L. H. S. Parekh, and D. M. Tilbury. Using mimo feedback control to enforce policies for interrelated metrics with application to the apache web server. In *Proc. of NOMS*, 2002.

[24] S. Duan and S. Babu. Processing forecasting queries. In *Proc. of VLDB*, 2007.

[25] S. Elnikety, E. M. Nahum, J. M. Tracey, and W. Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *Proc. of WWW*, 2004.

[26] R. J. Fowler, A. L. Cox, S. Elnikety, and W. Zwaenepoel. Using performance reflection in systems software. In *Proc. of HotOS*, 2003.

[27] N. Friedman, D. Geiger, and M. Goldszmidt. Bayesian network classifiers. *Machine Learning*, 29(2-3), 1997.

[28] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing performance isolation across virtual machines in xen. In *Middleware*, 2006.

[29] H.-U. Heiss and R. Wagner. Adaptive load control in transaction processing systems. In *Proc. of VLDB*. Morgan Kaufmann, 1991.

[30] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[31] J. Heo, X. Zhu, P. Padala, and Z. Wang. Memory overbooking and dynamic control of xen virtual machines in consolidated environments. In *IM*, 2009.

[32] Hyper-V server. http://www.microsoft.com/servers/hyper-v-server.

[33] E. Ipek, O. Mutlu, J. F. Martinez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA*, 2008.

[34] J.M.Blanquer, A.Batchelli, K.Schauser, and R.Wolsk. Quorum: Flexible quality of service for internet services. In G. M. Lohman, A. Sernadas, and R. Camps, editors, *Proc. of NSDI)*, 2005.

[35] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. In *ASPLOS*, 2006.

[36] C.-H. Jung, C.-S. Ham, and K.-I. Lee. A real-time self-tuning fuzzy controller through scaling factor adjustment for the steam generator of npp. *Fuzzy Sets Syst.*, 74, 1995.

[37] E. Kalyvianaki, T. Charalambous, and S. Hand. Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters. In *ICAC*, 2009.

[38] A. Kamra, V. Misra, and E. M. Nahum. Yaksha: a self-tuning controller for managing the performance of 3-tiered web sites. In *IWQoS*, 2004.

[39] A. Kansal, F. Zhao, J. Liu, N. Kothari, and A. A. Bhattacharya. Virtual machine power metering and provisioning. In *SOCC*, 2010.

[40] M. Karlsson, C. T. Karamanolis, and X. Zhu. Triage: performance isolation and differentiation for storage systems. In *IWQoS*, 2004.

[41] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proc. of IJCAI*, 1995.

[42] S. Kumar, V. Talwar, V. Kumar, P. Ranganathan, and K. Schwan. vmanage: loosely coupled platform and virtualization management in data centers. In *ICAC*, 2009.

[43] D. Kusic, J. O. Kephart, J. E. Hanson, N. Kandasamy, and G. Jiang. Power and performance management of virtualized computing environments via lookahead control. In *ICAC*, 2008.

[44] X. Liu, L. Sha, Y. Diao, S. Froehlich, J. L. Hellerstein, and S. S. Parekh. Online response time optimization of apache web server. In *IWQoS*, 2003.

[45] C. Lu, T. F. Abdelzaher, J. A. Stankovic, and S. H. Son. A feedback control approach for guaranteeing relative delays in web servers. In *RTAS*, 2001.

[46] D. Magenheimer. Memory overcommit...without the commitment. Technical report, Xen Summit, June 2009.

[47] J. C. Mogul. Emergent(mis) behavior vs. complex software systems. In *ACM SIGOPS Operating System Review*, 2006.

[48] MySql. http://www.mysql.com.

[49] D. Ongaro, A. L. Cox, and S. Rixner. Scheduling i/o in virtual machine monitors. In *VEE*, 2008.

[50] OProfile. http://oprofile.sourceforge.net/.

[51] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. In *EuroSys*, 2009.

[52] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *EuroSys*, 2007.

[53] PAPI. http://icl.cs.utk.edu/papi.

[54] PerfCtr. http://user.it.uu.se/ mikpe/linux/perfctr.

[55] PerfSuite. http://perfsuite.ncsa.uiuc.edu.

[56] J. Rao, X. Bu, C.-Z. Xu, and K. Wang. A distributed self-learning approach for elastic provisioning of virtualized cloud resources. In *MASCOTS*, 2011.

[57] J. Rao, X. Bu, C.-Z. Xu, L. Wang, and G. Yin. VCONF: a reinforcement learning approach to virtual machines auto-configuration. In *ICAC*, 2009.

[58] J. Rao, Y. Wei, J. Gong, and C.-Z. Xu. Dynaqos: Model-free self-tuning fuzzy control of virtualized resources for qos provisioning. In *IWQoS*, 2011.

[59] J. Rao and C.-Z. Xu. Online measurement the capacity of multi-tier websites using hardware performance counters. In *ICDCS*, 2008.

[60] P. P. Renu, P. Pradhan, R. Tewari, S. Sahu, A. Ch, and P. Shenoy. An observation-based approach towards self-managing web servers. In *IWQoS*, 2002.

[61] Rice University Computer Science Department. http://www.cs.rice.edu/CS/Systems/DynaServer.

[62] V. T. R.Iyer and K. Kant. Overload control mechanisms for web servers. In *Proceeding of Workshop on Performance and QoS of Next Generation Networks*, 2000.

[63] K. Shen, M. Zhong, S. Dwarkadas, C. Li, C. Stewart, and X. Zhang. Hardware counter driven on-the-fly request signatures. In *Proc. of ASPLOS*, 2008.

[64] F. G. Shinskey. *Process Control Systems: Application, Design, and Tuning.* McGraw-Hill, 1996.

[65] A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. In *SIGMOD Conference*, 2008.

[66] Y.-Y. Su, M. Attariyan, and J. Flinn. Autobash: improving configuration management with operating system causality analysis. In *SOSP*, 2007.

[67] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, 1998.

[68] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani. On the use of hybrid reinforcement learning for autonomic resource allocation. *Cluster Computing*, 2007.

[69] The ClarkNet Internet traffic trace. http://ita.ee.lbl.gov/html/contrib/ClarkNet-HTTP.html.

[70] The dm-ioband bandwidth controller. http://sourceforge.net/apps/trac/ioband/wiki/dm-ioband.

[71] The SPECweb benchmark. http://www.spec.org/web2005.

[72] The Transaction Processing Council (TPC). http://www.tpc.org/tpcw.

[73] The Transaction Processing Council (TPC). http://www.tpc.org/tpcw.

[74] The Transaction Processing Council (TPC). http://www.tpc.org/tpcc.

[75] Tomcat. http://tomcat.apache.org/.

[76] VMware. http://www.vmware.com.

[77] VMware VMmark. http://www.vmware.com/products/vmmark.

[78] E. Walker. Bechmarking Amazon EC2 for high-performance scientific computing. *;LOGIN:, the USENIX Magazine*, 2008.

[79] R. Wang, D. M. Kusic, and N. Kandasamy. A distributed control framework for performance management of virtualized computing environments. In *ICAC*, 2010.

[80] J. Wei and C.-Z. Xu. A self-tuning fuzzy control approach for end-to-end qos guarantees in web servers. In *IWQoS*, 2005.

[81] J. Wei and C.-Z. Xu. eqos: Provisioning of client-perceived end-to-end qos guarantees in web servers. *IEEE Trans. Computers*, 55(12), 2006.

[82] J. Wei and C.-Z. Xu. eqos: Provisioning of client-perceived end-to-end qos guarantees in web servers. *IEEE Transaction on Computer*, 55, 2006.

[83] WEKA. http://www.cs.waikato.ac.nz/ml/weka.

[84] M. Welsh and D. E. Culler. Adaptive overload control for busy internet servers. In *Proc. of USITS*, 2003.

[85] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *OSDI*, 2004.

[86] J. Wildstrom, P. Stone, E. Witchel, and M. Dahlin. Machine learning for on-line hardware reconfiguration. In *Proc. of IJCAI*, 2007.

[87] Xen. http://www.xen.org/.

[88] C.-Z. Xu. *Scalable and Secure Internet Services and Architecture*. Chapman and Hall/CRC Press, 2005.

[89] T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proc. of ISCA*, 1992.

[90] S. Zhang, I. Cohen, M. Goldszmidt, J. Symons, and A. Fox. Ensembles of models for automated diagnosis of system performance problems. In *Proc. of DSN*, 2005.

[91] S. Zhang, I. Cohen, M. Goldszmidt, J. Symons, and A. Fox. Ensembles of models for automated diagnosis of system performance problems. In *DSN*, 2005.

[92] W. Zhao and Z. Wang. Dynamic memory balancing for virtual machines. In *VEE*, 2009.

[93] X. Zhong and C.-Z. Xu. Energy-aware modeling and scheduling for dynamic voltage scaling with statistical real-time guarantee. *IEEE Trans. on Computers*, 56(3), 2007.

[94] F. Zhou, M. Goel, P. Desnoyers, and R. Sundaram. Scheduler vulnerabilities and attacks in cloud computing. *arXiv:1103.0759v1 [cs.DC]*, Mar 2011.

[95] X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, P. Padala, and K. Shin. What does control theory bring to systems research? *SIGOPS Oper. Syst. Rev.*, 43, 2009.

[96] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS'10*, 2010.

ABSTRACT

AUTONOMIC MANAGEMENT OF VIRTUALIZED RESOURCES IN CLOUD COMPUTING

by

JIA RAO

December 2011

Advisor:	Dr. Cheng-Zhong Xu
Major:	Computer Engineering
Degree:	Doctor of Philosophy

he last five years have witnessed a rapid growth of cloud computing in business, governmental and educational IT deployment. The success of cloud services depends critically on the effective management of virtualized resources. A key requirement of cloud management is the ability to dynamically match resource allocations to actual demands, To this end, we aim to design and implement a cloud resource management mechanism that manages underlying complexity, automates resource provisioning and controls client-perceived quality of service (QoS) while still achieving resource efficiency.

The design of an automatic resource management centers on two questions: when to adjust resource allocations and how much to adjust. In a cloud, applications have different definitions on capacity and cloud dynamics makes it difficult to determine a static resource to performance relationship. In this dissertation, we have proposed a generic metric that measures application capacity, designed model-independent and adaptive approaches to manage resources and built a cloud management system scalable to a cluster of machines.

To understand web system capacity, we propose to use a metric of productivity

index (PI), which is defined as the ratio of yield to cost, to measure the system processing capability online. PI is a generic concept that can be applied to different levels to monitor system progress in order to identify if more capacity is needed. We applied the concept of PI to the problem of overload prevention in multi-tier websites. The overload predictor built on the PI metric shows more accurate and responsive overload prevention compared to conventional approaches.

To address the issue of the lack of accurate server model, we propose a model-independent fuzzy control based approach for CPU allocation. For adaptive and stable control performance, we embed the controller with self-tuning output amplification and flexible rule selection. Finally, we build a QoS provisioning framework that supports multi-objective QoS control and service differentiation. Experiments on a virtual cluster with two service classes show the effectiveness of our approach in both performance and power control.

To address the problems of complex interplay between resources and process delays in fine-grained multi-resource allocation, we consider capacity management as a decision-making problem and employ reinforcement learning (RL) to optimize the process. The optimization depends on the trial-and-error interactions with the cloud system. In order to improve the initial management performance, we propose a model-based RL algorithm. The neural network based environment model, which is learned from previous management history, generates simulated resource allocations for the RL agent. Experiment results on heterogeneous applications show that our approach makes efficient use of limited interactions and find near optimal resource configurations within 7 steps.

Finally, we present a distributed reinforcement learning approach to the cluster-wide cloud resource management. We decompose the cluster-wide resource allocation problem into sub-problems concerning individual VM resource configurations. The cluster-wide allocation is optimized if individual VMs meet their SLA with a high resource utilization. For scalability, we develop an efficient reinforcement learning approach with continuous state space. For adaptability, we use VM low-level runtime statistics to accommodate workload dynamics. Prototyped in a iBalloon system, the distributed learning approach successfully manages 128 VMs on a 16-node closely correlated cluster.

AUTOBIOGRAPHICAL STATEMENT

JIA RAO

Jia Rao is a graduate student of Department of Electrical and Computer Engineering at Wayne State University. He received his B.S. and M.S. degrees in computer science and technology from Wuhan University, China in 2004 and 2006, respectively.

His research interests include cloud computing, virtualization, resource management, mobile computing, machine learning, reinforcement learning, CPU and I/O scheduling in virtualized systems. He has published 1 journal paper, in IEEE Transactions on Parallel and Distributed Systems, and 8 papers in proceedings of leading international conferences. He is also a receipient of the *Summer Dissertation Fellowship* (2011).

He has taught undergraduate courses for four years at Wayne State University. He has also received outstanding evaluations each semester from his students.