MULTI-DIMENSIONAL OPTIMIZATION FOR CLOUD BASED MULTI-TIER APPLICATIONS

A Thesis
Presented to
The Academic Faculty

by

Gueyoung Jung

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

Georgia Institute of Technology
Dec. 2010

MULTI-DIMENSIONAL OPTIMIZATION FOR CLOUD BASED MULTI-TIER APPLICATIONS

Approved by:

Professor Calton Pu, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Richard Schlichting
AT&T Labs Research

Professor Ling Liu
School of Computer Science
Georgia Institute of Technology

Professor Karsten Schwan
School of Computer Science
Georgia Institute of Technology

Professor Xue Liu
Department of Computer Science & Engineering
University of Nebraska Lincoln

Professor Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: Oct. 28, 2010

To my family,

Suyeul and Eden,

Father and Mother

ACKNOWLEDGEMENTS

On a long journey of my Ph.D program, I've met many people who deserve the credit. Without their help, this dissertation would not have been possible. First and foremost, I have been greatly privileged to work with an exceptional academic advisor, Prof. Calton Pu. Calton welcomed me into his research group of excellent students from early years in my Ph.D. program. Calton has always been a great mentor not only academically but also personally and financially. Calton has supported me to mainly focus on my research over my time in Elba group. More importantly, Calton has patiently believed me, inspired me, and guided me to the way to become a good scholar through the rest of my life. I am truly grateful to my advisor for helping me to build my own vision which would grow to be broader and deeper over time.

I would like to thank Dr. Rick Schlichting, Dr. Matti Hiltunen, and Dr. Kaustubh Joshi who have worked with me since I've joined their research group as an intern in Summer 2007. They have been great research collaborators over the last three years. Working with them has been a pleasure during summer internships at AT&T Labs Research. While working with them, I could always count on their sound technical feedback ranging from building systems to editing papers. Specifically, Rick encouraged me to find ways to solve them and do good research although he did not solve the problems for me. As one of my thesis committee members, Rick also gave me his insightful comments on this dissertation. Matti and Kaustubh have shared their experiences and thoughts with me during the various stages of my research that have shaped the course of this dissertation. I am really grateful to them for the investment they made for me.

I also thank my thesis committee members, Prof. Ling Liu, Prof. Karsten Schwan, Prof. Sudhakar Yalamanchili, and Prof. Xue Liu, who shared their valuable time to discuss

my work and research directions. Their constructive suggestions and questions have really helped me to improve this dissertation as a more concrete work. Their technical feedback led me to further consider how practically my work can be applied to real world issues.

I thank my friends, Galen Swint, Lenin Singaravelu, Jinpeng Wei, Younggyun Koh, and Qinyi Wu who helped me to get through the ups and downs of the roller coaster ride of Ph.D. program. I also thank the members of the Elba group for their friendship. I have enjoyed working with these talented colleagues and shared all adventures of Ph.D. program together from tedious daily happenings to discussion on researches. Simon Malkowski, Deepal Jayasinghe, Junhee Park, Pengcheng Xiong, and Qinyang Wang deserve a special mention.

Additionally, I thank Dr. Akhil Sahai who shaped my early research direction in the Elba group, and Dr. Ravi Konuru and Dr. Lionel Villard who hired me as an intern in IBM T.J. Watson Research Center. I would also like to thank Korea government, Ministry of Information and Communication, for supporting me in the first four years of my Ph.D. program. Their financial support allowed me to focus on my own research.

Most of all, I was able to finish this dissertation since I have been given the greatest and deepest support from my family. Their encouragement and love have always left me inspired. My wife, Suyeul, deserves a special mention. I specially thank for her unbelievable patience for such a long time, and her emotional support and encouragement was always a source of energy to me to rebound from a setback.

Finally, I thank God for being acquainted with these people on a long journey of my life. I believe that you are always beside me and strengthen me whenever I'm in vain or in pain.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Emerging trends toward cloud computing and virtualization have been opening new avenues to meet enormous demands of space, resource utilization, and energy efficiency in modern data centers. By being allowed to host many multi-tier applications in consolidated environments, cloud infrastructure providers enable resources to be shared among these applications at a very fine granularity. Meanwhile, resource virtualization has recently gained considerable attention in the design of computer systems and become a key ingredient for cloud computing. It provides significant improvement of aggregated power efficiency and high resource utilization by enabling resource consolidation. It also allows infrastructure providers to manage their resources in an agile way under highly dynamic conditions.

However, these trends also raise significant challenges to researchers and practitioners to successfully achieve agile resource management in consolidated environments. First, they must deal with very different responsiveness and performance requirements of different applications, while handling dynamic changes in resource demands as applications' workloads change over time. Second, when provisioning resources, they must consider management costs such as power consumption and adaptation overheads (i.e., overheads incurred by dynamically reconfiguring resources). Dynamic provisioning of virtual resources entails the inherent performance-power tradeoff. Moreover, indiscriminate adaptations can result in significant overheads on power consumption and end-to-end performance. Hence, to achieve agile resource management, it is important to thoroughly investigate various performance characteristics of deployed applications, precisely integrate costs caused by adaptations, and then balance benefits and costs. Fundamentally, the research question is how to dynamically provision available resources for all deployed applications to maximize overall utility under time-varying workloads, while considering such management costs.

Given the scope of the problem space, this dissertation aims to develop an optimization system that not only meets performance requirements of deployed applications, but also addresses tradeoffs between performance, power consumption, and adaptation overheads. To fulfill the goal, first, I have studied performance characteristics of enterprise multi-tier applications, and then built an adaptation engine to optimize end-to-end performance through dynamic resource provisioning techniques in a consolidated server environment. Second, I have investigated the impact of adaptation overheads on end-to-end response time and then integrated such transient adaptation costs into the adaptation engine to balance performance benefit and cost by developing analytical models and a novel optimization search algorithm.

This dissertation makes two distinct contributions. First, I show that adaptations applied to cloud infrastructures can cause significant overheads on not only end-to-end response time, but also server power consumption. Moreover, I show that such costs can vary in intensity and time scale against workload, adaptation types, and performance characteristics of hosted applications. Second, I address multi-dimensional optimization between server power consumption, performance benefit, and transient costs incurred by various adaptations. Additionally, I incorporate the overhead of the optimization procedure itself into the problem formulation. Typically, system optimization approaches entail intensive computations and potentially have a long delay to deal with a huge search space in cloud computing infrastructures. Therefore, this type of cost cannot be ignored when adaptation plans are designed. In this multi-dimensional optimization work, scalable optimization algorithm and hierarchical adaptation architecture are developed to handle many applications, hosting servers, and various adaptations to support various time-scale adaptation decisions.

CHAPTER I

INTRODUCTION

A key issue in the adaptive and autonomic computing vision is the automation of managing large application systems and IT infrastructures to serve millions of users with satisfactory performance. With the advent of cloud computing, today's enterprise computing resources and applications are more distributed in data center environments and more dynamic through on-demand utility computing under rapidly changing conditions than ever before. Additionally, various management objectives such as performance benefit, power saving, service availability, and management costs are increasingly inter-related and often conflicted. The growing complexity demands the automation of optimizing the utility of IT infrastructures and addressing various tradeoffs among such management objectives. Given these challenges, the goal of this dissertation is to develop rigorous and practical solutions to enable the multi-dimensional optimization of such complex systems.

1.1 Cloud Computing and Virtualization

Cloud computing is revolutionizing the computing landscape by making unprecedented levels of computing services cheaply available to millions of users. Recently, platforms such as Amazon's Elastic Compute Cloud (EC2) [3], AT&T's Synaptic Hosting [6], Google's App Engine [33], and Salesforce's Force.com [65] host a variety of distributed online applications including multi-tier e-commerce, social networking services, email, and CRM. Cloud computing platforms provide the collection of these services to end users based on negotiated service-level agreements (SLAs). To meet quality of service requirements specified in SLAs, such platforms involve the provision of dynamically scalable and often virtualized resources for hosted applications as a service - so called "Infrastructure as a Service." As the major advantage of cloud computing, cloud infrastructure providers can

potentially achieve the significant improvement of aggregated power efficiency and high resource utilization by densely packing hosted applications into a small number of physical machines. Meanwhile, infrastructure customers can amortize the cost of ownership and the cost of computing resource management, since they can avoid capital expenditure by renting the physical infrastructure for their services from cloud infrastructure providers. Altogether, cloud computing has become one of the most important future computing paradigms [80, 10, 53].

Virtualization technology such as Xen [7], VMware [76], and Microsoft Hyper-V [55] has recently gained considerable attention in the design of computer systems and data centers mainly due to its capabilities of the server consolidation, secure isolation between multiple virtualized machines, and agile resource management. Virtualization allows a single physical machine to be shared among multiple operating platforms called virtual machines (VMs) in secure isolation from each other. It also enables dynamic resource provisioning by allowing infrastructure providers to control and adapt the capacities of resources such as CPU, memory, and disk space at a very fine granularity. By this means, they can allocate such resources to applications only as needed and not statically allocate based on the peak workload demand. Moreover, it provides server consolidation facilities through VM migration and resource capping techniques. By consolidating multiple online applications onto fewer resources, infrastructure providers can achieve higher resource utilization while maintaining the desired quality requirements of hosted applications. Consequently, the virtualized infrastructure is considered as an essential building block to leverage the emerging cloud computing paradigms [10, 53].

1.2 Research Challenges

Although cloud computing built around virtualization is opening new avenues to meet enormous demands of high resource utilization and power savings in modern data centers, it also raises significant challenges. First, the sharing of resources by multiple applications raises

new resource management challenges such as ensuring responsiveness requirements of individual applications under dynamically changing workloads, and isolating applications from demand fluctuations in co-located VMs. Despite the well-documented importance of the responsiveness requirement to end users [29, 13, 79], today's cloud services typically address only availability guarantees and not SLAs based on end-to-end response times of hosted applications. Additionally, the multi-tier architecture style of hosted applications [1] makes the resource management more challenging. Virtualization technology provides additional flexibility to the scaling of applications by allowing the dynamic VM replication to host additional instances of applications' tier components. However, tier components of each application typically have different resource demands and strong dependencies each other for the application's overall performance [88, 74].

Second, dynamic consolidation and provisioning of virtual resources for hosted applications entails an inherent performance-power tradeoff [26, 47, 56, 70, 34, 12]. By facilitating server consolidation technique (i.e., dynamically packing applications/VMs into a small number of physical machines and shutting down idle machines), infrastructure providers can achieve significant energy efficiency. However, consolidation can also have a detrimental impact on the application performance. Hence, it must be used very carefully in a wide array of response time-sensitive applications such as online shopping, communications, and enterprise applications, where savings cannot come at the cost of a degraded user experience.

Third, the tradeoff between adaptation benefit and cost must be considered in the resource management, since runtime adaptation actions such as VM migration and replication entail performance degradation and additional power consumption for short periods [18, 83, 75, 32, 51, 77, 50]. While virtualization technology has made great strides in reducing the downtime during migration to a few hundred milliseconds (e.g., [20]), the

[1]The multi-tier architecture is a client-server architecture in which the presentation, the application processing, and the data management are logically separate processes, and they are typically distributed over the cluster or data center environments.

end-to-end performance and power consumption impacts can still be significant. In addition to these adaptation overheads, the power consumption and decision delay incurred by the resource management system itself must also be considered - so called "consuming power to save power." Typically, system optimization approaches entail intensive computations and could have a long delay to deal with a huge search space. Therefore, this type of cost cannot be ignored when adaptation plans are designed. Altogether, to achieve advantages of cloud platforms, infrastructure providers must account for not only balancing steady-state performance and power consumption, but also balancing steady state performance and power with overheads incurred by dynamic adaptations and decision making procedure itself under changing workloads.

Finally, the scalability of the dynamic resource management must be considered. To choose optimal adaptation actions to transform the current system configuration to an optimal one, the resource management system must explore a number of candidate configurations by applying all possible resource allocations and VM placements to available physical hosting machines for all hosted applications. Then, the optimization problem may show an exponential increase in worst-case complexity with the rapid growth of cloud infrastructures in scale, in terms of hosting machines/VMs, hosted applications, and various time-scale adaptation actions. Hence, the size of the search space poses a challenge for centralized implementations of the resource management system to provide fast adaptation decisions.

1.3 Thesis Statement

Given these challenges, the thesis of this dissertation is that the adaptation system through dynamic resource provisioning can address the multi-dimensional optimization problem inherent in maximizing adaptation benefits while minimizing management costs under rapidly changing environments such as cloud computing.

1.4 Solution Approach

To this end, we have thoroughly studied the performance characteristics and responsiveness of enterprise multi-tier applications, and used results for bottleneck detection [45, 68, 61, 44]. Then, an adaptation system has been developed to optimize end-to-end response time through dynamic resource provisioning in a consolidated server environment [39, 41, 40]. Recently, we have investigated the impact of adaptation overheads on end-to-end response time, and then integrated the VM migration cost into the adaptation system by developing system models and a novel optimization search algorithm [42].

In this dissertation, we show that adaptations can cause significant overheads not only on end-to-end performance but also on power consumption. Moreover, such costs can vary in intensity and time scale against workload, adaptation types, and the performance characteristics of hosted applications and their tier components. Then, the adaptation system is extended to solve the multi-dimensional optimization problem that balances power consumption, application performance, and accrued power/performance costs incurred over various adaptation actions and decision making in a single unified framework.

To achieve the multi-dimensional optimization, we have developed various models and optimization methods. First, model-based prediction is adopted. Using analytical modeling techniques, the adaptation system can predict the application's end-to-end response time, CPU utilization, and power consumption for a given workload. Models enable the adaptation system to capture virtualization overhead and the responsiveness of the multi-tier application that has unique features such as the synchronous resource possession between tiers. Model construction is based on a layered queueing network model template. Model parameters are determined by measuring the application off-line. This measurement is fully automated, and does not require application knowledge or modification of the application code. We also construct cost models that quantify the degradation in the application performance and the additional power consumption incurred by adaptation actions using the automatic off-line experimentation. Second, a utility function is used to convert predicted

performance benefit, power consumption, and accrued costs over adaptations to utility values and then, balance them on a uniform footing. Finally, by developing a scalable search algorithm and adaptation architecture, we can efficiently choose optimal adaptation actions to maximize the overall utility for multiple multi-tier applications under dynamic workload conditions. The implemented approach is evaluated in a small data center setup. Especially, we evaluate the multi-dimensional optimization approach by performing extensive experimental comparisons with three alternative strategies, each of which represents optimizing the tradeoff between any two objectives among performance, power consumption, and transient costs.

1.5 Important Technical Contributions

To achieve this multi-dimensional optimization, this dissertation makes the following unique contributions.

First, our approach optimizes the application performance in a transparent and tractable way. Since typical online controllers make adaptation decisions algorithmically using complicated stochastic models, they often lack transparency and predictability. Our adaptation system provides a mechanism that constructs adaptation policies off-line for a given range of workload as a rule set. The rule set can be used as a guideline for online adaptation decisions by offering the upper-bound system configuration for a given condition. The generated rule set is represented in the form of replication levels and resource capacities of all participating applications. Additionally, such a rule set provides the enhancement of system manageability, since it is human readable and extensible. This allows, for instance, further inspection or modification with additional constraints such as the consideration of adaptation overheads by domain experts.

Second, our approach precisely incorporates adaptation overheads into the optimization formulation. While virtualization makes reconfiguration easy, our research efforts demonstrate that the indiscriminate use of adaptations such as VM replication and VM migration

can have significant impacts on the ability to satisfy response-time-based SLAs and power savings. By developing a utility function to formulate the problem of the performance-aware, power-aware, and cost-aware resource provisioning, we can address not only the tradeoff between performance and power consumption at steady-state, but also balance accrued adaptation costs and their benefits to maximize the overall utility.

Finally, our approach solves the multi-dimensional optimization problem in an efficient way. Our adaptation system is designed to balance multiple management objectives including performance, power consumption, and transient adaptation costs incurred by adaptation actions and the decision making procedure itself. To balance the system optimality and the cost of decision making, a scalable, self-aware optimization algorithm is developed. Specifically, a best-first graph search algorithm based on the models and utility function is developed to choose optimal sequences of adaptation actions. Additionally, a multi-level hierarchical adaptation architecture is developed to deal with a large number of applications and hosts, and also with various adaptation decisions at multiple time-scales ranging from a few milliseconds to hours.

1.6 Organization of This Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 introduces optimization formulations and model-based prediction techniques that are developed for this research. Chapter 3 introduces an off-line approach developed to optimize the application performance, and shows some representative evaluation results. Chapter 4 describes the cost-sensitive performance optimization that integrates adaptation overheads into the on-line decision making process, and show evaluation results. Chapter 5 introduces some motivated observations and discusses the solution for multi-dimensional optimization problem and evaluation results in detail. Finally, Chapter 6 outlines the contributions of this dissertation research and future directions of this research work.

CHAPTER II

BACKGROUND AND FOUNDATION

In this chapter, first, background of this dissertation is introduced. Specifically, we present the implications of a multi-tier application architecture on dynamic resource provisioning, and then introduce the cloud infrastructure used for the evaluation of our optimization solutions. Subsequently, an optimization formulation is described in the form of a utility function. The adaptation system is designed to optimize over dual objectives of power and performance, and therefore, it uses a *utility* based model to compare both on a uniform footing. Finally, a set of system modeling methodologies are described. Since the adaptation system is based on predictive adaptations, it involves model-based prediction methodologies. These modeling methodologies include the queuing models used to predict the application performance, the analytical models used to predict the overall system's power consumption, the measurement-based techniques used to predict transient adaptation costs, and the predictive filter used to estimate how long the system workload will remain approximately at its current status.

2.1 Multi-Tier Applications

In this dissertation, a shared resource pool hosting multiple multi-tier enterprise applications is considered. While the multi-tier architecture style has become an industry standard in modern data centers, the separation of application into multiple tiers makes resource management under changing workload conditions more challenging. A multi-tier application typically consists of a Web server (e.g., Apache), an application container (e.g., PHP, Java-based application servers), and a backend database (e.g., MySQL, Oracle). Specifically, resources for tiers have to be provisioned separately since each tier has different

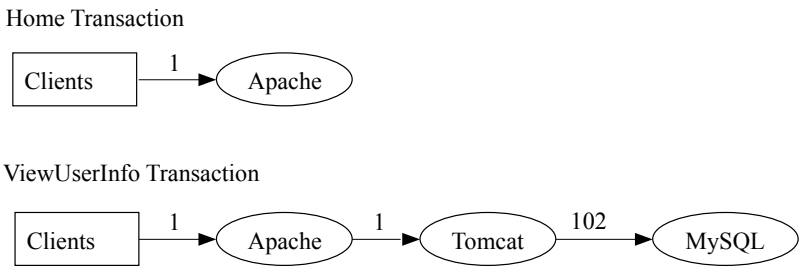


Figure 1: Behaviors of two representative transaction types in a multi-tier benchmark (RUBiS)

performance characteristics and thus, different resource demands. Additionally, the performance bottleneck often shifts from one tier to another as the workload changes over time [88, 74]. For instance, increasing CPU allocation for the application server tier due to an observation of resource shortage at the tier may merely shift the bottleneck to the database tier. Therefore, the adaptation mechanism must balance resource demands efficiently among tiers as well as among the different hosted applications to optimize overall performance.

The workload of such applications is complex as well. The number of concurrent sessions may change over time, and a client session to these applications consists of different types of transactions with potentially very different characteristics and resource requirements. More specifically, different transactions utilize different subsets of the tiers and the interactions between tiers caused by a transaction may vary greatly as illustrated in Figure 1. This example shows two different types of transactions called "Home" and "ViewUserInfo" of RUBiS benchmark that emulates an online auction system with a 3-tier architecture consisting of Apache Web server, Tomcat application container, and MySQL database server. The "Home" transaction involves only a call to Apache to request a static HTML page but for the "ViewUserInfo" transaction, Apache makes a single call to Tomcat, which in turn makes an average of 102 calls to MySQL.

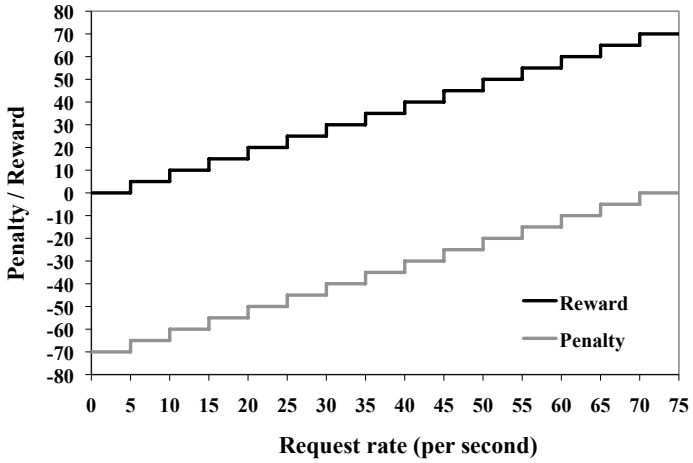


Figure 2: SLA-based utility

Each application has an SLA that provides a mapping from the quality of service provided by an infrastructure provider to a reward or penalty owed to or from the infrastructure provider, respectively. A simple pricing scheme is employed in this research, in which the infrastructure customer (i.e., the application owner) pays the infrastructure provider when the end-to-end mean response time is smaller than a threshold (i.e., a target response time), and the infrastructure provider pays the infrastructure customer when the mean response time is larger than the threshold. Each hosted application may have its own rewards, penalties, and response time thresholds. We assume that the target response time can be derived experimentally as the mean response time across all transactions of a single hosted application running in isolation in the initial configuration driven by a constant workload equal to half of the design workload. Furthermore, it is assumed that rewards and penalties are affected by workload (i.e., request rates) as illustrated in Figure 2. For example, as the workload of an application increases, the infrastructure provider should receive a higher reward or a lower penalty to compensate for the increased amount of work. Finally, we assume that the maximum request rate of each application is specified in its SLA, and that the system has an admission control mechanism capable of enforcing this maximum request rate. This makes it possible to ensure the load never increases beyond anticipated workloads. However, the infrastructure provider has a financial incentive to host as many

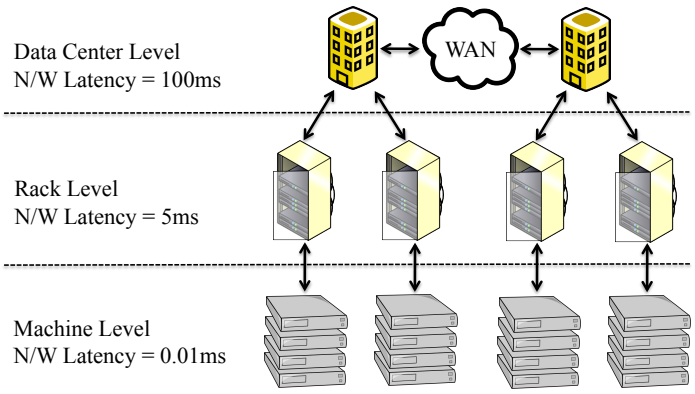


Figure 3: Resource hierarchy of a cloud infrastructure

applications as possible to maximize its revenue.

2.2 Cloud Computing Infrastructure

In this dissertation, a consolidated data center environment is considered, in which a set of multi-tier applications are to be deployed across a set of physical hosting machines located across multiple data centers. Each hosting machine in our environment is virtualized with Xen [7]. As is typical in such environments, we assume that the hosting machines are organized into hierarchical groupings such as racks, clusters, and data centers to facilitate networking and management. These groupings are represented by a resource hierarchy that includes a set of "resource levels" (e.g., machine, rack, data center levels), and a hosting relation between them (e.g., a data center hosts multiple racks, each of which hosts multiple machines). Figure 3 shows an example resource hierarchy that includes three levels with 16 machines distributed across four racks in two data centers. Hosting machines are assumed to be inter-connected by a data center network. We also assume that the network does not need to be implemented in any particular way, but inter-machine latency increases as one moves higher up the resource hierarchy. For example, machines placed in the same rack have a lower network latency between them than machines across different racks, which have a lower latency than machines distributed across different data centers.

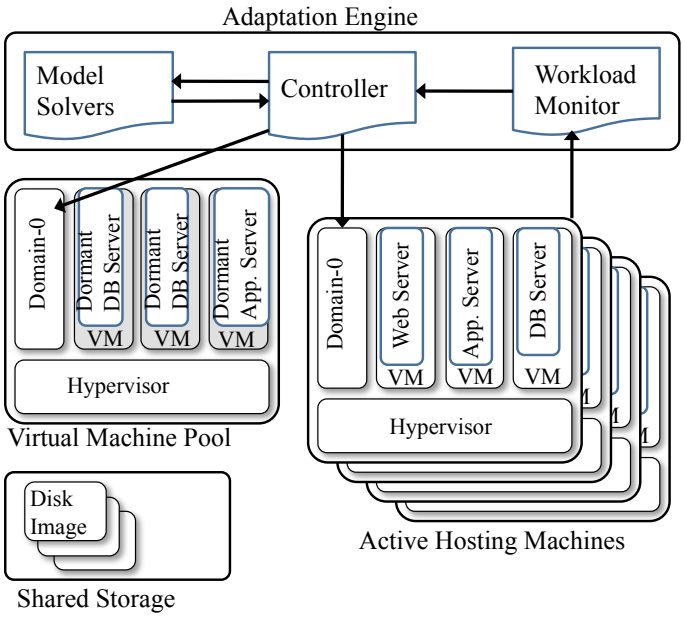


Figure 4: Resource group as the management unit

Given the resource hierarchy, the adaptation system is designed to deploy multiple instances of controllers in the form of a multi-level hierarchical control scheme. Each controller manages different subsets of hosting machines by communicating model solvers and workload monitor and then generating a set of adaptation actions. The lowest level controllers manage a small number of machines (e.g., a single rack). At the next higher level, a controller manages machines owned by multiple lower level controllers. The complete control architecture is described in Chapter 5.

Figure 4 illustrates one of the low resource groups that consists of a cluster of commodity machines inter-connected by a local Ethernet in our Xen-based virtualization test-bed. Several machines are used to actively host a set of multi-tier applications. Tier components of each application can be distributed across physical machines and VMs. As shown in the figure, we assume that each VM executes a single application tier component (e.g., Web, application, or database server) at any given time for the sake of convenience. The controller is deployed on a separate machine with a workload monitor. At runtime, the controller periodically reads current workloads from the workload monitor and generates a

set of adaptation actions such as increase/decrease the CPU allocation of a VM, change the replication level of a tier component, and migrate a tier component from one physical machine to another. These actions are triggered by the distinguished VM, called Domain-0, on each Xen-based hosting machine that manages the other VMs. One machine in the resource group is dedicated to hosting dormant VMs used in server replication. The replication level of an application component is increased by migrating a dormant VM from the VM pool to the target hosting machine, allocating it a CPU capacity, configuring the application, and starting the VM. Similarly, the replication level is decreased by deactivating the VM and migrating it back to the VM pool. Finally, as is common in data center environments, a shared storage network is used, so all VMs on the resource group use disk images stored on the same shared storage machine.

2.3 Optimization Formulation: Utility Functions

To describe our utility formulations, we begin by formally defining $configuration$. A pool of computing resources $\mathcal{R}$ and a set of multi-tier applications S are considered. For each application $s \in S$, let N^s be the set of its constituent component types (e.g., Web server, application server, and database server), and for each component type $n \in N^s$, let $\text{repset}(n)$ be a set of allowed replication levels. Let N_k^s denote the set of nodes in the system, where a node may be a component (for non-replicated components) or a component replica. For example, a 3-tier application consisting of Apache with up to 2 replicas, Tomcat with up to 3 replicas, and an unreplicated MySQL database server has $\text{repset}(\text{apache}) = \{1, 2\}$, $\text{repset}(\text{tomcat}) = \{1, 2, 3\}$, and $\text{repset}(\text{mysql}) = \{1\}$. If Tomcat is replicated twice in a particular configuration, then the set of nodes $N_k^s = \{\text{apache1}, \text{tomcat1}, \text{tomcat2}, \text{mysql1}\}$. Each application s may also support multiple transaction types $T^s = \{t_1^s, \ \ldots, \ t_{|T^s|}^s\}$. The workload for the application can then be characterized by the set of request rates for each of its transactions, or $w^s = \{w_t^s | t \in T^s\}$, and the workload for the entire system by $W = \{w^s | s \in S\}$.

Given this, the goal of an adaptation system is to configure the target system such that for a given workload W, the utility U of the entire system is maximized. This maximization is carried out over the space of all possible system configurations C, where each $c \in C$ specifies: (a) the replication level $c.\mathrm{rep}(n)$ of each node n of each application s from the set $\mathrm{repset}(n)$, (b) the assignment of each replica $n_k \in N_k^s$ to a physical resource $c.\mathrm{r}(n_k)$, and (c) the fractional allocation (i.e.,capacity) $c.\mathrm{cap}(n_k) \in [0, 1]$ of the resource. A constraint applied to the definition is that the sum of the fractional allocations across all nodes of all applications in a single hosting machine is at most 1 for each resource.

To decide $when$ and how to adapt configuration at runtime, the adaptation system estimates the potential benefit of each adaptation action a as well as its cost in terms of changes in power and performance utility values. The cost of each adaptation action a depends on its duration $d(a)$ and impact on response time and power consumption. Then, the total cumulative cost is the sum of costs incurred by all adaptation actions applied to transform the current configuration into a new configuration. Meanwhile, the total benefit of adaptation actions depends on how long the system remains in the new configuration. Thus, the overall *system utility* U consists of the *power utility* U_{pwr} in the new configuration, the sum of individual application s's *performance utilities* U_{RT}^s based on the end-to-end response time in the new configuration, and the sum of individual action a's transient *adaptation cost* including performance degradation $U_{RT}(c, a)$ and additional power consumption $U_{pwr}(c, a)$.

To compute performance utility, each application has its own performance objective in the form of a target mean response time $TRT^s(w^s)$ computed over fixed monitoring window M, a reward $R(w^s)$ for meeting the target response time in a single monitoring period, and a penalty $P(w^s)$ for missing it. The response time targets, rewards, and penalties are defined to depend on the request rate, thus arbitrary application utility functions are allowed to be defined. As described in Chapter 2.1, our adaptation system uses a function in which the reward increases and the penalty decreases as the workload increases. Given the measured or predicted request rate for application s at time-step i as W_i^s, the system

configuration as c_i, the measured or predicted mean response time RT_i^s, the target response time $TRT^s(W_i^s)$, reward $R^s(W_i^s)$, and penalty $P^s(W_i^s)$, the application utility accrual rate is given as:

$$U^s_{RT_i}(c_i) = \begin{cases} R^s(W_i^s)/M & \text{if } RT_i^s \leq TRT^s \\ P^s(W_i^s)/M & \text{otherwise} \end{cases} \quad (1)$$

The total performance utility for the new configuration is then simply the sum of performance utility values of all applications.

To incorporate the performance degradation as a adaptation cost factor into the optimization formulation, it is converted to a utility value. The adaptation system computes the instantaneous rate at which an application accrues utility during the execution of a series of adaptation actions in an interval. Let $RT^s(c_i, a)$ be the measured or predicted mean response time of application s of the system, when adaptation action a is executed in configuration c_i. By plugging this value into Equation 1, the corresponding utility accrual rate $U^s_{RT_i}(c_i, a)$ during execution of action a starting from a configuration c_i can be computed. Then, the adaptation system puts together these components to obtain the overall utility accrued between two invocations of a controller. Let the initial configuration be c_i, and let CW_{i-1} be the stability interval as defined earlier. Let W_i^s represent the fixed workload during the stability interval. The stability interval ends when the workload deviates from a band of width B^s, called the *workload band*, centered around this fixed value (i.e., $(W_i^s - B^s/2, W_i^s + B^s/2)$). When that happens, the controller is invoked to evaluate the need for adaptation and may execute a sequence of adaptation actions $A_i = a_1, a_2, a_3, \ldots a_n$ to transform c_i into a new configuration c_{i+1}. We anticipate that this new configuration is retained until the end of the new stability interval CW_i. Let $d(a_1), d(a_2), \ldots, d(a_n)$ be the length of each adaptation action, and let $c^1, c^2, \ldots, c^n$ be intermediate configurations generated by applying the actions starting from c_i. Let c^0 be the initial configuration c_i and c^n be the final configuration c_{i+1}. Then, the overall utility at time i is given by

$$U_i = \sum_{a_k \in A_i} d(a_k) \cdot \big(\sum_{s \in S} U^s_{RT_i}(c^{k-1}, a_k) \big) + \big(CW_i - \sum_{a_k \in A_i} d(a_k) \big) \big(\sum_{s \in S} U^s_{RT_i}(c_{i+1}) \big) \quad (2)$$

The first term of the equation sums up the utility accrued by each application during each action in the adaptation sequence over a period equal to its action length, and second term sums the utility of the resulting configuration over the remainder of the control interval.

For the (negative) utility accrued due to power consumption, in this dissertation, the adaptation system focuses on power consumed by the physical hosts. While power consumed by cooling infrastructure is also a major concern in typical data centers, this adaptation system does not consider it explicitly since cooling overheads can be approximately modeled as a fixed percentage of the power consumed by the computing infrastructure [25]. We convert the energy cost per Watts-hour $PCWh$ to the instantaneous rate at which utility is accrued using the equation

$$U_{pwr}(c_i) = -pwr(c_i) \cdot PCWh \quad (3)$$

where $pwr(c_i)$ is the predicted or measured mean power consumption (in Watts) of the system in configuration c_i over the monitoring interval M.

Similar with the performance degradation above, the power consumption incurred by an adaptation can be converted to a utility value as following. The adaptation system computes the instantaneous rate at which an application accrues utility during the execution of a series of adaptation actions in an interval. Let $pwr(c_i, a)$ be the predicted power consumption of the system, when adaptation action a is executed in configuration c_i. The corresponding power utility accrual rate $U_{pwr}(c_i, a)$ incurred by action a from c_i can be obtained by applying the value into Equation 3. Finally, the adaptation system plugs the power utilities accrued both over adaptations and during the rest of the interval into Equation 2. Then, the overall utility is given by

$$U_i = \sum_{a_k \in A_i} d(a_k) \cdot \big(U_{pwr}(c^{k-1}, a_k) + \sum_{s \in S} U^s_{RT_i}(c^{k-1}, a_k)\big)$$
$$+ \big(CW_i - \sum_{a_k \in A_i} d(a_k)\big)\big(U_{pwr}(c^{i+1}) + \sum_{s \in S} U^s_{RT_i}(c_{i+1})\big) \quad (4)$$

The first term sums the system-wide power utility and application specific performance utilities accrued during each adaptation action execution (i.e., the action costs), while the second term sums the power and application utilities of the resulting configuration c_{i+1} until the end of the stability interval. By maximizing this utility, the adaptation system can balance the cost accrued over the duration of an adaptation with the benefit accrued between its completion and the next adaptation.

2.4 Model-Based Prediction

2.4.1 Application Modeling: Queueing Network Models

To estimate the benefit of a configuration, the end-to-end response time of each application needs to be estimated for a given workload. In this dissertation research, layered queuing network modeling (LQNM) techniques [84] are adopted.

Queueing models have been used for modeling multi-tier applications in a number of research projects including [59, 72, 73, 69], but unlike these efforts we choose LQNM. The primary reason is that in consolidated server environments with fine-grained CPU control and multiple applications, models need to be accurate over a wide range of workloads, high resource utilization, and even in configurations that might be very unbalanced in terms of resource allocation among tiers. Thus, a blocking phenomenon must be explicitly modeled although the phenomenon is insignificant in well-provisioned environments such as a bottleneck due to the blocking of front-end server threads by a highly overloaded back-end server. Unlike standard queuing models, layered queuing networks enable such modeling by allowing multiple resources to be consumed by a request at the same time. Such extended queuing networks can be solved through a number of algorithms based on mean

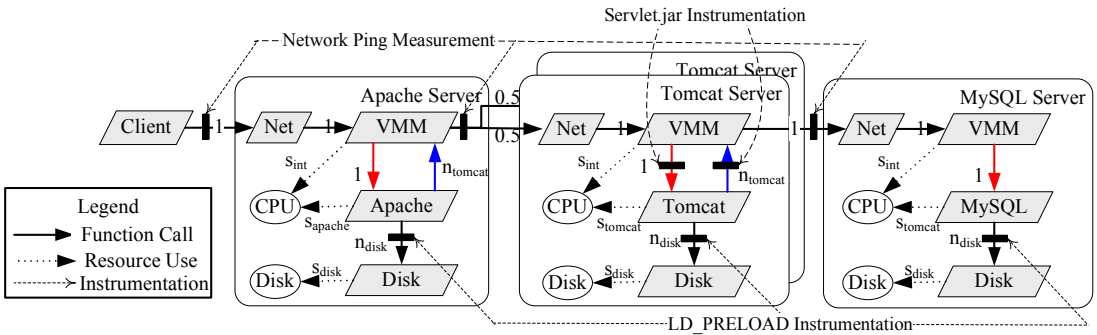


Figure 5: Layered queueing network model for 3-tier RUBiS benchmark

value analysis (MVA) (e.g., [38]). The LQNS modeling tool introduced by [28] is used as a black-box model solver.

An additional complication with the model is accounting for the overhead imposed by a virtualization architecture that allows hosted VMs to share resources. In particular, since Xen places device drivers for physical devices into a designated guest VM, referred to as Domain-0, all incoming and outgoing network communications pass through this extra domain and incur additional latency. Moreover, since Domain-0 shares the CPU with the other VMs, this latency depends on both the CPU utilization and the number of messages. This additional step is an intrinsic problem with virtualization techniques, and although system level methods to alleviate this problem have been proposed recently in (e.g., [35]), the problem is still an open research issue. Therefore, we explicitly model this VM monitor delay. Note that if virtualization overhead is not modeled explicitly, CPU utilization and response time predictions will be inaccurate.

A high-level diagram of the resulting model for a single example application is shown in Figure 5, and a more detailed portion of the model is shown in Figure 6 for two example transaction types that comprise the benchmark application. In the figures, the layered queuing models are specified in terms of tasks formed by software components (depicted by parallelograms) and queues that are formed by the hardware resources (depicted by circles) that are used by the tasks. When tasks use hardware resources (depicted by dotted

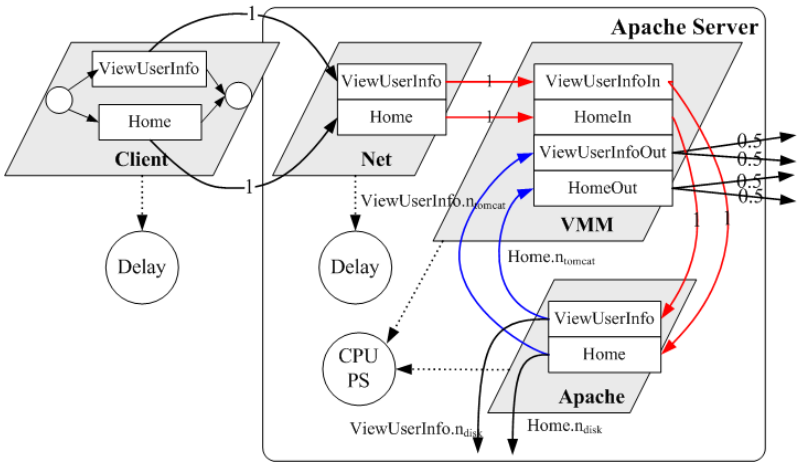


Figure 6: Detailed partial view of LQN model

arrows with the label specifying the service time), or when they make synchronous calls to other tasks (depicted by solid arrows with the label specifying the average number of calls made), both the caller and the callee servers are blocked. Finally, as the detailed model shows, each task comprises of a number of entries (depicted as rectangles), each of which correspond to a single transaction type in the system. These entries actually place demands on resources and make calls to other entries. Therefore, they are associated with parameters for service time and mean number of calls made to other entries. Note that the figures sometimes show only a single parameter value for all the entries in a task for brevity.

A pre-deployment training phase is used to collect all the parameters required by the model. This phase is fully automated and does not require any modification of the application code. During this phase, each application is measured in isolation with only a single component per tier, and is subjected to a workload that consists of a single transaction at a time. Multiple repetitions are done to compute mean values of the parameters. The process is then repeated for each transaction and in two environments, a virtualized environment in which each tier executes in its own VM and a native environment where each tier is given its own native OS without virtualization. The solid black boxes in the figures represent the points where measurements are made. A description of each task and how its parameters

are computed using these measurements is as follows.

Net represents the latency introduced by the network. Since the network is not assumed as a bottleneck resource, it is modeled as a pure delay server (i.e., no resource sharing). The service time is measured using `ICMP` ping measurements in the native environment across the resource hierarchy.

Disk represents the delay due to disk I/O. To measure the service time transparently, each application tier is wrapped with an interception library using the `LD_PRELOAD` environment variable. The library intercepts each `read` and `write` call made by the application to compute the mean number of I/O calls n_{disk} and their service time.

Component (e.g., Apache, Tomcat, and MySQL) represents the processing performed by the tier component. The task is modeled using an M/M/n queue, where n is set to the maximum number of software threads for each component (or 1 in the case of MySQL, which creates threads on demand). The threads execute on a CPU queue with the processor sharing discipline to approximate time-slice OS scheduling. To measure the service time of the Java-based application server transparently, we use binary rewriting to instrument the `Servlet.jar` file that is used by every application based on Java servlets. The instrumentation timestamps each incoming request from and response to the Web server. Logs are used to measure the service times of Web and database servers. The logging module of MySQL is modified to measure query service time and the number of calls per transaction. In addition, the client measures end-to-end response time for the entire transaction. Performing the experiment with only a single user at a time ensures that no queuing delay is present in the system, and the measurements at each server can be correlated.

Virtual Machine Monitor (VMM) represents the interaction delay induced by the virtualization environment. The service time for this task is assumed to be equal across all machines since it is dependent on virtualization layer and not on the application. To estimate this time, we first compute the difference between the service times of each tier in the virtualization environment with the VMM task, and in the native environment without the

VMM task. Then, using knowledge of the measurement points and how many times the VMM is included in each measurement, the VMM service time can be computed.

Client generates the workload for the queuing model. Since our controller uses the measured rates of individual transaction types at runtime, we use the workload model proposed in [88] that allows the typical Markov transition workload matrices to be converted into vectors of stationary probabilities of each transaction type. Using this approach, the workload for each application s can be modeled as a set of $|T^s|$ independent open Poisson processes.

2.4.2 Power Consumption Modeling

To estimate power consumption of a configuration c for a given workload W, a utilization-based power model is adopted that has been validated in previous studies (e.g., [25, 50]). Specifically, for a physical machine, an empirical non-linear model is used as following,

$$pwr = pwr_{idle} + (pwr_{busy} - pwr_{idle}) * (2\rho - \rho^r) \tag{5}$$

where pwr_{idle} represents the power consumption of the machine at standby state and pwr_{busy} represents the maximum power consumption of the physical machine observed in our workload scenarios, and ρ is the CPU utilization of the machine estimated by the LQN models at the current workload. A tuning parameter r is used to minimize the square error. It can be obtained at a model calibration phase. Through offline experiments, the non-linear model is calibrated to fit into actual power consumption observed using a power meter. The total power usage of the system is simply the sum of all participating physical machines' power usages including active VMs and shared domains (i.e., Domain-0s).

In Chapter 5.3.2, we will illustrate the power model fitting process and its accuracy with a real world world.

2.4.3 Transient Adaptation Costs

In this dissertation work, various time-varying adaptation actions are considered. They include increase/decrease a VM's CPU capacity by a fixed amount, addition/removal of a

VM containing an application tier's replica, live-migration of a VM between hosting machines located across the resource hierarchy, shutting down/restarting physical machines, and disk image migration. Addition of a VM replica is implemented by migrating a dormant VM from a pool of VMs to the target hosting machine and activating it by allocating CPU capacity. A replica is removed by migrating it back to the standby pool. Some actions also require additional coordination in other tiers. For example, changing the replication degree of the application server tier requires updating the front-end Web servers with new membership.

Our approach for the cost prediction is based on approximate models. Costs of these adaptation actions are measured experimentally off-line for different workloads and configurations and are stored in a mapping tables used at runtime. Specifically, the measurement is conducted for the following attributes of each adaptation: (a) adaptation duration, (b) change in response time for the application being adapted as well as applications co-located with the application being adapted, and (c) change in power consumption during the adaptation.

The following process to measure these costs is used. For each application s, workload w^s, and adaptation action a, we set up the target application along with a background application s' such that all replicas from both applications are allocated equal CPU capacity (40% in our experiments). Then, we run multiple experiments, each with a random placement of all VMs across all the physical hosting machines. During each experiment, we subject both the target and background application to the workload w^s and $w^{s'}$, and after a warm-up period of 1 minute, measure response times of two applications RT^s and $RT^{s'}$ and the total power usage of corresponding physical machines pwr. Then, we execute the adaptation action a, and measure the duration of the action, $d(a)$, the response time of each application during adaptation, $RT^s(a)$ and $RT^{s'}(a)$, and the power usage on affected physical machines $pwr(a)$. If none of application's VMs are co-located with the VM impacted by

a, no background application measurements are made. We use these measurements to calculate delta response times $\Delta RT^s(a) = RT^s(a) - RT^s$ and $\Delta RT^{s'}(a) = RT^{s'}(a) - RT^{s'}$, and the delta power usage $\Delta pwr(a) = pwr(a) - pwr$. These deltas along with the action duration are averaged across all random configurations, and their values are encoded in a cost table indexed by the workload.

When a controller requires an estimate of adaptation costs at runtime, it measures the current workload W and looks up the cost table entry with the closest workload W'. To determine the impact of the adaptation action a on its target application s, it measures the current response time of the application as RT^s and estimates the new response time during adaptation as $RT^s(a) = RT^s + \Delta RT^s(a)$. For each application s' whose components are hosted on the same machine targeted by a, it calculates the new response times as $RT^{s'}(a) = RT^{s'} + \Delta RT^{s'}(a)$. The power consumption can be calculated with similar way.

Although this technique does not capture fine-grained variations due to the difference between configurations or workloads, we will show that the estimates are sufficiently accurate for making good decisions.

2.4.4 Workload Stability Prediction: ARMA Filter

Given that our approach balances immediate adaptation costs versus their potential future benefits, the ability to estimate workload changes is crucial. The approach we have chosen is to estimate how long the workload stays approximately at its current level based on the history of workload changes. The stability interval for an application s at time t is the period of time for which its workload remains within $\pm b/2$ of the measured workload W_t^s at time t, where b is a user-specified threshold. This band [W_t^s - $b/2$, W_t^s + $b/2$] is called the *workload band* B_t^s. When an application's workload exceeds the workload band, the controller must re-evaluate the system configuration. When the workload falls below the band, the controller must check if other applications might benefit from the resources that could be freed up.

At each unit monitoring interval i, the controller checks if the current workload W_i^s is within the current workload band B_j^s. If one or more application workloads are not within their band, the estimator estimates a new stability interval CW_{j+1}^e for the next control window and updates the bands based on the current application workloads. To estimate the stability intervals, we employ an auto-regressive moving average (ARMA) filter commonly used for time-series analysis (e.g. [9]). The filter uses a combination of the last measured stability interval CW_j^m and an average of the k previously measured stability intervals to predict the next stability interval using the equation:

$$CW_{j+1}^e = (1-\beta) \cdot CW_j^m + \beta \cdot 1/k \sum_{i=1}^{k} CW_{j-i}^m \tag{6}$$

Here, the factor β determines how much the estimator weighs the current measurement against past historical measurements. It is calculated using an adaptive filter to quickly respond to large changes in the stability interval while remaining robust against small variations. To calculate β, the estimator first calculates the error ε_j between the current stability interval measurement CW_j^m and the estimation CW_j^e using both current measurements and the previous k error values as,

$$\varepsilon_j = (1-\gamma) \cdot |CW_j^e - CW_j^m| + \gamma \cdot 1/k \sum_{i=1}^{k} \varepsilon_{j-i} \tag{7}$$

Then, β of Equation 6 can be computed by $1 - \varepsilon_j / \max_{i=0\ldots k} \varepsilon_{j-i}$. This technique dynamically gives more weight to the current stability interval measurement by generating a low value for β when the estimated stability interval at time i is close to the measured value. Otherwise, it increases β to emphasize past history.

2.5 Summary

In this chapter, first, we have introduced the environment, where our approach is applied. The architecture of the multi-tier application and its complexity have been introduced.

Then, a cloud infrastructure as the form of hierarchical resource structure has been presented. We consider that multiple multi-tier applications are deployed across resource levels of the hierarchical structure. Under the dynamic changes of applications' workloads, this research aims to optimize the overall utility of cloud infrastructures by provisioning resources. In order to identify an optimized configuration for a given workload, we should estimate the utility of each possible configuration. We have described a utility function that contains power consumption, performance benefit obtained by adaptations and cost incurred by adaptations. Finally, we have introduced analytical modeling techniques that are integrated into the utility function. The autoregressive moving average technique has been presented. It is used to estimate the period that the current workload will remain approximately as its current status. In each period, we should estimate each adaptation's cost, and the final configuration's performance and power consumption. We have introduced the queuing network models used to predict the application performance, the non-linear utilization based power model used to predict the overall system's power consumption, and the measurement-based techniques used to predict transient adaptation cost. The accuracy of these models described in this chapter is shown in the following three chapters.

CHAPTER III

PERFORMANCE OPTIMIZATION

As the first part of this dissertation research, an off-line approach has been developed to optimize the application performance in a Xen-based virtualized environment that hosts multiple multi-tier applications. By constructing adaptation policies off-line, this approach can support online policy-driven adaptation systems to efficiently and transparently provision available resources under time-varying workloads. This approach supports the multi-dimensional adaptation system by providing the upper-bound performance utility that is used in the multi-dimensional optimization process.

3.1 Problem Statement

A management challenge in data center environments that host distributed multi-tier applications is dealing with rapidly changing execution conditions such as time-varying application workloads. The ideal solution to this problem is to reallocate resources dynamically at a fine grained level using *adaptive* or *autonomic system* techniques, yet support for creating adaptation policies that guide when and how to change allocations is often lacking. The challenge in this environment is even more severe given the wide range of application types that have to be supported and the complex nature of multi-tier applications.

The recent trend towards consolidated server environments based on virtualized resources provides a unique opportunity to enhance the autonomic management capabilities of data centers hosting such applications. It becomes easier, for example, to allocate resources to a given component of a multi-tier application by changing the percentage of the CPU allocated to VM executing that component or by replicating the VM onto a second physical machine. Virtualized environments also enable the sharing of resources across applications in ways not previously possible, which potentially allows higher utilization

of resources such as CPU and memory. The fundamental issue, then, is how to change the allocation of resources to applications dynamically to optimize the overall performance under changing conditions, while considering resource availability and application-specific characteristics such as SLAs and workloads.

3.1.1 Resource Provisioning: On-line vs. Off-line

Many approaches have been proposed for dynamically adapting system behavior to optimize resource usage and application performance, including techniques based on stochastic models (e.g., [27, 8, 72, 22, 71, 88]), reinforcement learning (e.g., [69]), and control theory (e.g., [59, 58, 57]). Although the details are different, each follows a similar pattern: construct a parametric model of the target system (e.g., a queuing model); fix some model parameters through experimental measurement or learning; devise a strategy for optimizing the remaining parameters using the runtime state as input; implement the strategy in an online controller that is periodically provided with the measured runtime system state; and use the recommendations of the controller to adapt the system configuration.

The disadvantage of online controllers is that by generating decisions algorithmically and only on demand, they may give rise to undesirable emergent properties, impede the ability of administrators to understand system behaviors, and ultimately, reduce the predictability of the target system. While some techniques - most notably control-theoretic ones - attempt to remedy concerns of unpredictable and undesired behaviors by proving stability properties of the control algorithms, by doing so they limit the system models in significant ways (e.g., through linearity assumptions) or run the risk that the guarantees are invalidated if the assumptions are not met. In contrast, rule-based expert systems address autonomic management using rules written by domain experts and executed using engines such as HP Openview [36] and IBM Tivoli [37]. Unlike online approaches, the use of pre-determined rule bases provides predictability, but with the drawback that the rules are often hard to write and cumbersome to maintain given their tight linkage to the underlying

system.

In this work, we have proposed a novel hybrid approach for enabling autonomic behavior that provides the best of both worlds. It uses a utility function and an analytical modeling technique along with optimization algorithm to predict system performance and automatically generate optimal system configurations. Rather than producing these configurations on demand at runtime, they are produced off-line to feed a decision-tree learner that produces a compact rule set (or *adaptation policy*). This rule set can be directly used in rule engines, audited, combined with other human-produced rules, or simply used to aid domain experts in writing and maintaining management policies. This approach of producing entire decision rule sets off-line has another benefit as well - the modeling solution and optimization is entirely removed from the critical path of the system during runtime. Therefore, it is possible to model and optimize ever larger and more complex systems. The utility function and queueing network models introduced in Chapter 2 are used as the prediction models.

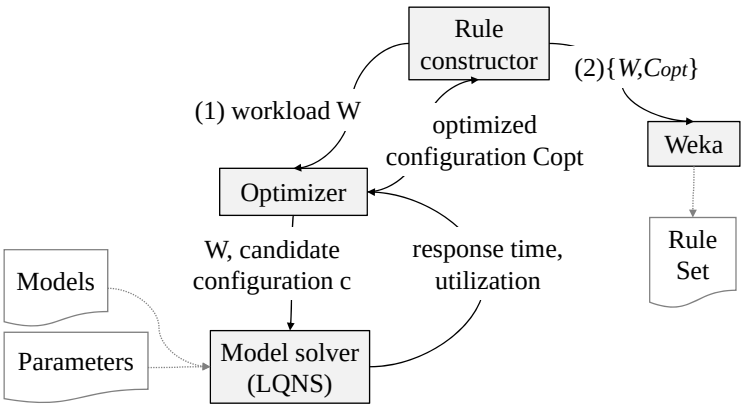


Figure 7: Policy generation approach

3.2 Approach

3.2.1 Overview: Rule Set Generation

Our off-line approach for automatically generating adaptation rule sets is outlined in Figure 7. A queueing model $m_a \in M$ for each application $a \in A$ has been developed, and these models are fed into the model solver. Using the model solver, the optimizer generates a raw rule set, which is a list of pairs matching a condition predicate with the corresponding target configuration. To determine a point in the raw rule set, the rule set constructor randomly selects a workload (i.e., a set of request rates) within workload ranges specified in SLAs and passes it to the optimizer. Given this workload, the optimizer determines an optimized configuration by searching over the possible set of candidate configurations until it finds the best (or near-optimal) configuration. For each candidate configuration, it uses the model solver to compute a set of expected mean response times of each transaction type and utilization of each component of the applications. This process runs until it collects enough entries for the raw rule set. Finally, the rule constructor generates an "if-then-else" policy rule set from the raw rule set using a decision tree learner of Weka tool.

3.2.2 Performance Optimizer

The configuration optimizer uses the queueing models described in Chapter 2.4.1 and the performance utility function described in Equation 1 in Chapter 2.3 to search for a system configuration that maximizes the overall performance utility for a given workload. This task includes replication level selection and CPU allocation for all components, and placement of components on physical hosts. Due to the extremely large configuration space involved and the fact that the queuing models do not have a closed form solution, the optimization task is computationally challenging. It is easy to show that the problem is NP-Complete by a reduction to the bin-packing problem and thus, an exact solution is not possible. Furthermore, even an approximate solution based on gradient search is not sufficient due to the discrete input space in component placement.

To solve the problem in an efficient manner, we split it into two sub-problems: (a) selecting for each application an *application configuration* consisting of the replication level and CPU allocation for each component, and (b) mapping the components determined by the application configuration onto the physical machines. For each candidate set of optimal application configurations, the component placement acts as an accept-reject mechanism. If the placement step can fit the required components into the available resources, then the application configurations are accepted. Otherwise, they are rejected. The optimization algorithm is shown in Algorithm 1 and explained in detail below.

Input: W - the workload at which to optimize
Output: c_{opt} - the optimized configuration
forall $a \in A, n \in N_a$ **do**
  $c.\text{rep}(n) \leftarrow \max\{\text{reps}(n)\}, \forall n_k, c.\text{cap}(n_k) \leftarrow 1$
forall $a \in A$ **do**
  $(RT_a, \{\rho(n_k)|\forall n_k \in N_a^k\}) \leftarrow$ `LQNS` (W, a, c)
Compute U
while *forever* **do**
  $\{c.r(n_k)|\forall a, n_k\} \leftarrow$ `BinPack` $(R, \{\rho(n_k)\})$
  if *success* **then return** c
  foreach $a \in A, n \in N_a$ **do**
    $c^r \leftarrow c[\text{rep}(n) \leftarrow$ Next smallest in $\text{reps}(n)]$
    $C^k \leftarrow \{c[\text{cap}(n_k) \leftarrow$ Reduce by $\Delta r]|\forall k\}$
    foreach $c_{new} \in \{c^r\} \cup C^k$ **do**
      $(RT_a, \{\rho(n_k)\})_{new} \leftarrow$ `LQNS` (W, a, c_{new})
      Compute U_{new}, $\nabla\rho$
      if $\nabla\rho < 0 \vee \nabla\rho$ *is max so far* **then**
        $(c, \{\rho(n)\})_{opt} \leftarrow (c, \{\rho(n)\})_{new}$
      if $\nabla\rho < 0$ **then** skip to **EndRefit**
  EndRefit: $(c, \{\rho(n)\}) \leftarrow (c, \{\rho(n)\})_{opt}$

Algorithm 1: Optimal configuration generation

The optimization algorithm uses a discrete gradient-based search algorithm to explore candidate configurations. Note that (a) for any application and its transaction type, the utility function (i.e., Equation 1) is monotonically decreasing with increasing response time, (b) the response time monotonically (but not necessarily strictly) increases with a reduction in the number of replicas of a tier, and (c) the response time monotonically increases with a

reduction in the resource fraction allocated to the replicas of a tier. Hence, the utility function has its maximal value for a configuration where each tier is replicated to its highest allowed replication level and a full resource fraction (i.e., 1.0 per replica) is allocated to each component. Naturally, this configuration will typically not fit in the set of available physical resources and thus either replication levels and/or resource fractions have to be reduced until a fit is found.

Initially, the algorithm begins its search from this maximal utility configuration. For each application model, the LQNS solver is invoked to estimate response time and the actual CPU utilization $\rho(n_k)$ of the configuration of the application. The bin packer is then invoked to place all components onto the available hosts using the predicted CPU utilization as the "volume" of each component. If the bin packing is unsuccessful, the algorithm re-evaluates all possible single-change degradations of the current configuration by either reducing the replication level of a component to the next lower level or by reducing the CPU allocation of a component by a step of Δr. These reductions are done to each component in each application separately to generate a set of candidate configurations. Various granularities for the reduction of CPU capacity can be used (e.g., 1, 5, 10, or 15%), but a tradeoff between the optimality of result configuration and the efficiency of the algorithm must be considered. In particular, the algorithm terminates faster the larger the reduction granularity, but with an increased chance that it may skip a configuration that would have had a larger utility value than the chosen one. In our current implementation, we use 5% reduction step since it shows a good optimality with reasonable execution time. During the evaluation, only the model for the application with the reduced resources has to be solved again, resulting in computational savings. The algorithm then picks the degraded configuration that provides the maximum reduction in overall CPU utilization for a unit reduction in utility, or gradient, which is defined as:

$$\nabla\rho = \frac{\sum_{a\in A, n_k \in N_a^k} \rho_{\text{new}}(n_k) - \rho(n_k)}{U_{\text{new}} - U} \tag{8}$$

The whole process is repeated again until the bin packing succeeds. This technique is guaranteed to find a configuration that fits in the given resources since the CPU fractions allocated to components can be reduced repeatedly until the bin packing succeeds.

The component placement based on bin packing determines whether the candidate configurations fit into the available physical resources and then, determines the host assignment for each application component. Bin packing has been studied extensively in the literature, and many efficient algorithms can approximate the optimal solution within any fixed percentage of the optimality. In our implementation, we use the $n\ log\ n$ time first-fit decreasing algorithm that ensures results are asymptotically within 22.22% of the optimal solution (e.g., [21]).

Using the above techniques, the optimizer is able to generate an optimized configuration for a given workload. While this configuration is not provably optimal, experimental results demonstrate that the rule set that results from this process is effective and close to optimal (see Chapter 3.3).

The approach could also be extended to manage multiple heterogeneous resource types by extending the bin packing algorithm to generate component placements with additional constraints. In particular, to allow for resources with different capacities, any of several approximation algorithms for the variable sized bin-packing problem (e.g., [46]) could be used, while to incorporate additional resource types such as memory or network bandwidth, algorithms for the vector bin-packing problem (e.g., [17]) could be used. Algorithms for both of these extensions could be incorporated into our approach to generate component placements without modifying other aspects of our approach.

3.2.3 Off-line Rule Set Constructor

The rule set constructor generates the actual adaptation rules used by the on-line controller. Using the allowed range of request rates for each transaction type of each application, this module randomly generates a set of candidate workloads WS. For each

workload $W \in WS$, it invokes the optimizer to find the best configuration c^*. We encode c^* as a string consisting of physical host names followed by the list of components hosted on it. Each component entry indicates the name of the application, component name, and the CPU capacity allocated to it. For example, the notation "`host1 app1 apache1 60 app1 tomcat3 40`" indicates that `host1` hosts two components from `app1`: `apache1`, which is allocated 60% of the CPU, and `tomcat3`, which is allocated 40% of the CPU.

It is not feasible to evaluate every possible workload point to determine its optimal configuration. Therefore, we use the J48 decision tree learner of the Weka toolkit [81] to construct a decision tree as an interpolated, compacted rule set using the (W, c^*) pairs as the training data set. The generated decision tree has conditions of the form "$w_t^s <$ threshold" or "$w_t^s \geq$ threshold" at each of its branches, where w_t^s is the request rate for transaction t of application s. Each leaf of the tree encodes the configuration that should be used if all the conditions along the path are true, from the root to that leaf.

The decision tree construction serves multiple functions. First, it provides the interpolation that is needed for rule sets to be applicable not just for the points evaluated by the optimizer, but for any workload in the range allowed by the SLAs. Second, the decision tree can be trivially linearized into a nested "if-then-else" rule set that requires less expertise to understand than the models.

Figure 8 shows a fragment of the rule set used in our experiments. Third, due to the finite number of leaves in the decision tree, all configurations the system might include are known before deployment. This knowledge provides a degree of predictability and verifiability that is important for business-critical systems. Finally, the tree provides compaction of the raw training data since the learning algorithms aggregate portions that share similar configurations and prune outliers. As a consequence of compaction and due to the fact that at run-time, the system is likely to see a much larger set of points than were used to generate the tree, some loss of accuracy (and utility) is expected. In Chapter 3.3, we evaluate the

```
if (app0-Home > 0.023855)
    if(app1-Browse ≤ 0.175308)
        if (app0-BrowseRegions ≤ 0.05698)
            c = "host0app0mysql80" +
                "host1app1mysql80" +
                "host2app0tomcat40app1tomcat40" +
                "host3app0apache40app1apache40";
        if (app0-BrowseRegions > 0.05698)
            if (app1-Browse ≤ 0.119041)
                if (app1-Browse ≤ 0.086619)
                    c = "host0app0mysql80" +
                        "host1app0tomcat80" +
                        "host2app0apache60app1apache20" +
                        "host3app1mysql50app1tomcat30";
                if (app1-Browse > 0.086619)
                    c = "host0app0mysql80" +
                        "host1app0tomcat80" +
                        "host2app1mysql60app1apache20" +
                        "host3app1tomcat50app0apache30";
```

Figure 8: Rule set fragment generated by the off-line constructor

loss of accuracy and show that even with a modest number of training points, accurate rule sets can be constructed.

3.3 Evaluation Results

The goal of the evaluation is to demonstrate the feasibility and accuracy of the three steps in our approach: modeling, optimization, and rule set construction. Specifically, we show that (a) the constructed models accurately predict both response time and CPU utilization, (b) the optimizer produces configurations that are close to optimal, and (c) the resulting rule sets prescribe close to optimal configurations for any given workload.

3.3.1 Experimental Setup

RUBiS, a commonly used multi-tier application benchmark, is used in our evaluation scenario. RUBiS is an e-commerce application implementing the core features of an eBay-like auction site with 26 transaction types [14, 63]. RUBiS provides various user behaviors such

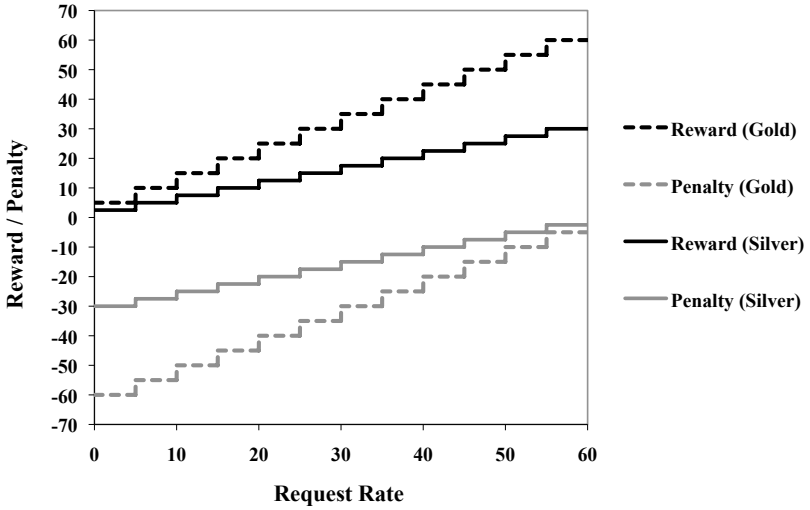


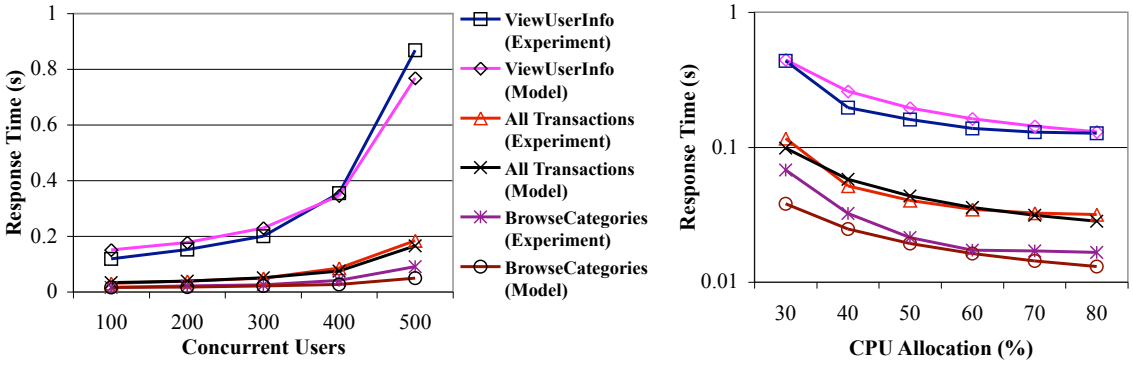
Figure 9: Step-wise pricing scheme

Table 1: Transaction types' thresholds

Transaction Type	Threshold (msec)
SearchItemsInCategory	170
SearchItemsInRegion	100
ViewBidHistory	340
ViewItem	100
ViewUserInfo	700
Browse	2
BrowseCategories	110
BrowseRegions	50
Home	2

as browsing, selling, and bidding for items. The benchmark defines two types of transition matrices representing read-only and read-write transactions. The read-only transaction matrix is used in our experiments. We deploy two types of RUBiS instances, referred to as gold and silver services. In this experiment, we assume each instance has its own SLA in terms of rewards and penalties, and then extend the SLA-based utility introduced in Section 2.1 as shown in Figure 9. We also assume that all RUBiS instances have the same response time thresholds as shown in Table 1.

To run instances of RUBiS, we have used four physical hosting machines each with



(a) Effect of Load (b) Effect of CPU Allocation

Figure 10: Response time of model vs. experimental results

an Intel Pentium 4 1.80GHz processor, 1 GB RAM, and a 100 Mb Ethernet interface. We have used the open-source version of the Xen 3.0.2 to build the virtualization environment. Linux kernel 2.6.16.29 has been installed as a guest OS in each domain of Xen. Apache 2.0.54, Tomcat 5.0.28, and MySQL 3.23.58 have been used as the Web server, servlet container, and database server respectively in 3-tier configurations of RUBiS. Each replica has been installed in its own VM. The concurrency parameter `maxClient` for the Apache servers has been set to 335 and `maxThreads` for the Tomcat servers set to 655 to avoid unnecessary thread induced blocking. We have increased the heap size of the Tomcat server to 512 MB to avoid slowdowns induced by garbage collection and enabled `db_connection_pool`. Finally, we have run the optimization process on a machine with 4 Intel Xeon 3.00GHz processors and 4 GB RAM.

3.3.2 Application Model Validation

Our approach requires that the models be accurate enough to predict both the end-to-end response time of the system and the CPU utilizations of the different system components with different workloads and different configurations. Figure 10 (a) demonstrates the accuracy of response time prediction for different transaction types and different workloads without replication and a CPU fraction of 55% for all components. The figure illustrates

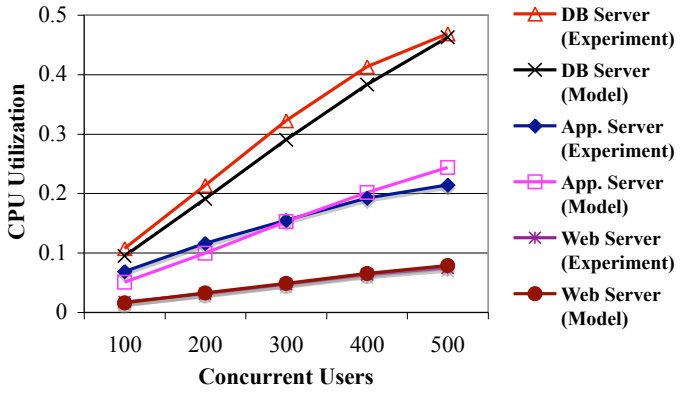


Figure 11: Utilization of model vs. experimental results

that the response times predicted by the model correspond well with the measured response times. Figure 10 (b) presents similar results when the CPU fractions of all components are adjusted from 30% to 80%. We set the workload (i.e., the number of concurrent users) to 200.

Figure 11 presents the predicted CPU utilization versus the measured CPU utilization at the three tiers as the workload increases. Each component runs over its own VM with a 55% CPU fraction and no replication has been used. Overall, these figures demonstrate that the models are reasonably accurate and can be used as the foundation for generating adaptation rules.

3.3.3 Accuracy of the Optimization Process

The optimization process employs a heuristic search algorithm that may miss the optimal configuration. In this section, we evaluate the accuracy of this search process using different parameters. Recall that the optimization process starts from a "maximal" configuration and reduces it by a certain CPU fraction or replication level at each step until a fit is found for the available resources. In our evaluation scenario, we set the maximal configuration to one replica for Apache, two replicas for Tomcat, and two replicas for MySQL, so that when deploying two applications, the total number of components is 10. For each replica, the initial CPU fraction is set to 80%, which is the maximum allowed by Xen since the

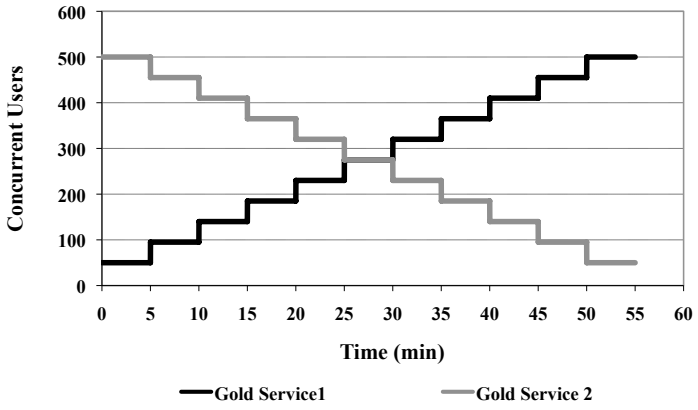
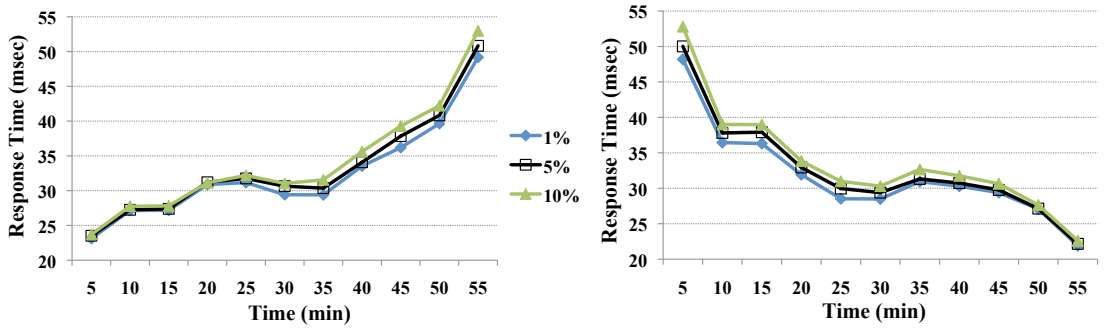


Figure 12: Simple workload scenario

remaining 20% is used by domain-0.

To evaluate the accuracy of the optimization process, we designed a simple scenario as shown in Figure 12, where two RUBiS instances referred to as GoldService1 and GoldService2 are deployed with identical SLAs. We then generate synthetic workloads within a defined load range in which the number of concurrent users is between 10 and 500 using client workload generators provided in RUBiS. The workload of GoldService1 starts at 50 and increases by 45 every 5 minutes until it reaches 500, while the workload of GoldService2 starts from 500 and decreases by 45 every 5 minutes. The rationale behind the scenario is that the optimizer has to allocate more resources to GoldService2 than GoldService1 during the first 5 minute round, and then must keep moving resources from GoldService2 to GoldService1 to optimize the overall configuration during the remainder of the rounds in the experiment. Note that, in this experiment, we restart the services for each round with the configuration determined by the optimizer to exclude any noise incurred by the adaptation process. Each round has a 1 minute warm-up phase followed by a 4 minute measurement phase.

With this scenario, two methods are used to evaluate the accuracy of the optimization process. First, we show the impact of varying the step sizes of the CPU reduction on the efficiency of the optimizer and the optimality of each resulting configuration. As mentioned



(a) GoldService1 (b) GoldService2

Figure 13: Response times of three different step sizes in the optimizer

Table 2: Execution time and accuracy.

	CPU Reduction Step		
	1%	5%	10%
Running Time (msec)	240694	50136	26875
Number of evaluated configurations	1152	235	121
Average Response Time (msec)	32.15	32.95	33.93
Difference from "1%" case (%)	-	2.56	5.39

in Chapter 3.2.2, when the search algorithm employs a small CPU reduction step size, the chance of missing the optimal configuration is reduced but the overall evaluation time increases since the optimizer has to evaluate more configurations. The efficiency of the optimizer is represented as "Running time" (i.e., how long it takes to obtain a target configuration) and "Number of evaluated configurations" (i.e., how many configurations the optimizer evaluates to obtain a target configuration), and the optimality of each resulting configuration by "Average response time" of the resulting configuration.

Table 2 summarizes the results of three different step sizes. The numbers in the table are average values over all rounds in the scenario. Figure 13 illustrates the distribution of each service's response time over the experiment. The results demonstrate that the difference in accuracy is negligible across the three different step sizes, although the execution time increases significantly as the step size is decreased. We use the 5% step size for the rest of our experiments since the result is both reasonably accurate and computationally efficient.

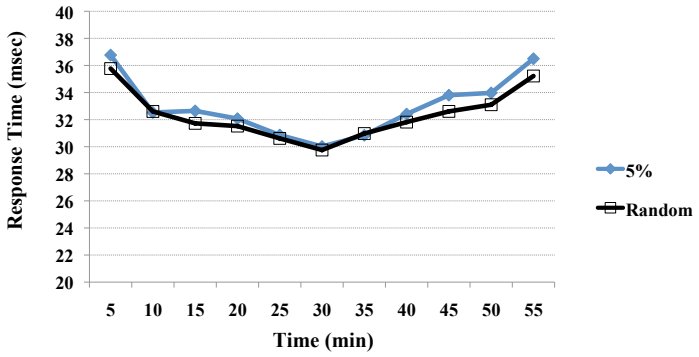


Figure 14: Global quality of the optimizer

Second, to evaluate the global optimality of the solution, we compare the configuration chosen by the optimizer with the best configuration from a large number of randomly generated configurations for a given workload. We generated 20000 configurations for each round in the scenario by randomly selecting each component's CPU fraction, replication level, and assignment to a host. Then, the configuration that minimizes the average response time of the two applications is determined. The CPU fraction space for the random configurations is discretized for fair comparison, specifically, the CPU fraction is randomly selected from among the values 30, 35, 40, 45, etc. In each round in the scenario, most of the random configurations perform very poorly, as expected, and only a few are close to or slightly better than the result of our optimizer.

Figure 14 shows that the response times of the configuration produced by the optimizer are very close to the best of the random configurations for each workload point in the scenario. In fact, the best random configuration is the same as the configuration selected by the optimizer for 5 of the rounds, and only slightly better in the other rounds. The average difference between response times of the two configurations is 1.89%, and the utilities are identical.

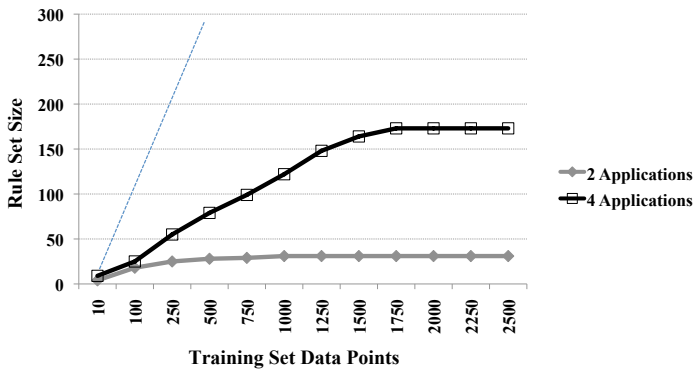


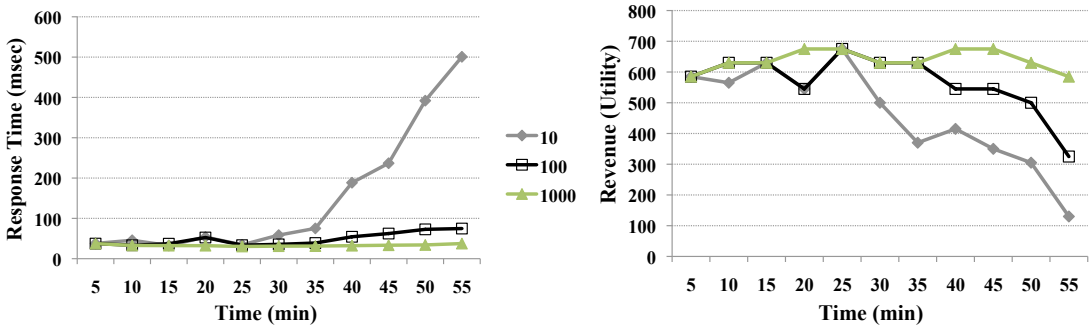
Figure 15: Size of rule set

3.3.4 Accuracy of the Constructed Rule Set

The goals of the rule set construction phase described in Chapter 3.2.3 are to minimize both the size of the resulting rule set and the loss of utility compared to the optimal configuration for any given workload. These goals may conflict. For example, we can reduce the size of the rule set by merging configurations that have similar workloads or reducing the number of training data points, but optimality may suffer. In practice, the goal is to find a number for the size of the training data set that balances rule set size with optimality.

Our first experiment shows how the size of the rule set changes when the number of training data points varies. The results are presented in Figure 15. The dotted line denotes the line where the size of the rule set would be the same as the number of training data points (i.e., x = y). The results indicate that the size of the rule set depends not only on the number of training data points, but also the number of applications and hence, the number of components and complexity of the configurations. However, since both curves flatten out at a certain point, we can use the knee of each curve as the optimal number of training points.

Our next experiment evaluates the impact of the rule set size on its accuracy by generating three different rule sets for the two applications in our simple scenario (Figure 12) using 10, 100, and 1000 training data points, respectively. The sizes of these generated rule sets were 4, 18, and 31, respectively, where the size of a rule set is defined as the number of



(a) Response time (b) Utility

Figure 16: Three different rule sets

leaves in the set. The rule set generated by applying 1000 training data points is the upper bound since it is the point where the curve in Figure 15 starts to converge to 31 for two deployed applications.

Figure 16 presents the impact of the rule set size on both the average response time and the utility. When 10 training data points are used to generate the rule set, both the response time and the utility are slightly degraded in rounds 2, 4, and 6 compared to the rule set based on 1000 training points. The gap increases starting in round 7 as the load of GoldService1 keeps increasing in the scenario, since the selected configurations do not have enough resources for the database server(s) to deal with some of the more expensive transactions (e.g., "ViewUserInfo"). However, increasing the number of training data points from 10 to just 100 significantly reduces the gaps in the utility and especially the response time.

3.4 Work Related to Performance Optimization

A number of recent efforts have addressed dynamic resource provisioning of enterprise applications. For example, [69] has proposed a reinforcement learning approach to resource allocation but only for coarse-level provisioning at the host level (i.e., no resource sharing among hosted applications) and for single-tier applications, while [15] has dealt with fine-grained resource allocation but only for single-tier systems that can be described using

closed form performance prediction equations. [72] has considered multi-tier applications and used queuing models similar to ours, but performed only coarse-grained provisioning at the host level and did not deal with cases where sufficient resources are not available. [86] has presented a two-level control model where local controllers use fuzzy control and continuous learning to determine new resource requirements given the new workload, and a global controller allocates requested resources so that the allocation maximizes profits. They have also considered multi-tier applications in a virtualized environment, but have assumed that all components of an application run in one virtual container (i.e., a single VM). This is less realistic and more coarse-grained than our approach, which only requires that each component runs in a separate VM. Perhaps closest to our approach is [8], which proposes a resource provisioning methodology using queuing models and a beam search to perform resource allocation in data centers. They have validated the approach using a simulated system. However, this work does not address fine-grained resource sharing, and it does not appear trivial to extend their optimization approach to do so. Moreover, all of the above approaches are based on on-line control and do not consider off-line policy generation with its many benefits.

Queueing models have often been used to estimate the performance characteristics of Web servers, but much of the work in this area has focused on single-tier systems and has been used in online solutions as a means to augment feedback control loops [67, 66, 11, 24]. In contrast, we have applied queueing models off-line and use them for more complex multi-tier applications. Recently, multi-tier applications have been modeled in [72, 19, 58], but without accounting for the simultaneous resource possession typically exhibited in multi-tier applications as discussed in Chapter 2.4.1. We have used layered queueing models to deal with simultaneous resource possession by modeling synchronous interactions between servers. [54] has employed layered queueing network models for EJB-based enterprise systems with the goal of manual capacity planning. However, these models have not considered virtualized environments and limited their applicability to environments in

which separate application silos with designated resources are used. Since virtualization has a performance impact on transaction-based applications as mentioned in [49] and [78], these models cannot be applied directly to virtualized environments without significant changes. [19] has dealt with static provisioning of multi-tier applications executing in a Xen-based environment and developed queuing models. However, the performance overhead of virtualization is not reflected in the models, but rather handled separately using extensive experimental measurement to ascertain the impact. For instance, service times of application components across many different CPU allocations are measured and then used to parameterize the approach. [58] has used black-box linear models for predicting CPU utilization in virtualized environments, but the models do not consider the performance impact of virtualization and so, the approach is also dependent on extensive experimentation. These experiment-based techniques are an alternative method for incorporating the performance impact of virtualization and have the advantage of requiring less knowledge about the system. However, they do not scale well as the number of applications and tiers increases.

Optimization problems arising in multi-tier enterprise systems have been intensively studied, but few of these efforts have focused on the problem of dynamic resource provisioning. For example, [87] has used an optimization technique to determine per-transaction service times for a queuing network when the times are not directly measurable. [85] has proposed efficient search algorithms and used them to determine what experiments must be conducted to choose appropriate application parameters. Our approach could also utilize such search-based methods. Finally, [22] has addressed the problem of dynamic resource allocation in virtualized platforms hosting multi-tier applications. This approach uses a capacity manager that is periodically executed and reassigns resources by evaluating a model consisting of multi-tier M/M/1 queues. This is an online technique, however, and does not have the benefits of our hybrid approach.

Finally, machine learning and especially decision trees have been used by several authors for learning autonomic behaviors. For instance, [71] has used these techniques to predict threshold points where a system is likely to fail its service level agreement obligations. However, most of this previous work has used decision trees in their traditional role of learning classifiers based on experimental data. We are not aware of any other work that has used decision trees to generate adaptation or other management policies.

3.5 Summary

Constructing optimal adaptation policies is one of the biggest challenges in building complex autonomic systems. This research work has presented a novel hybrid approach for the automatic generation of such adaptation policies that uses a combination of offline model evaluation and optimization. A policy is generated as a rule set that is compact enough for human administrators to inspect and augment, and can be used directly with existing rule-based management engines. Experimental results using RUBiS demonstrate the viability of the approach for this type of common multi-tier application. Specifically, we have showed that the models used in the approach and the generated rule sets are accurate in consolidated server environments that our optimization heuristic identifies near-optimal configurations, including replication levels, component placement on physical hosts, and VM parameters tuned for dynamic workloads. All these results suggest that a hybrid approach such as the one presented here can be an effective starting point for automatically managing multi-tier enterprise applications in consolidated server environments.

CHAPTER IV

COST-SENSITIVE ADAPTATION

As another part of the dissertation, an online adaptation system has been developed to automatically generate adaptation actions to transform a current configuration to an optimal configuration under dynamic workload changes over time. In particular, this work has emphasized that the cost of VM migration is significant and then, precisely incorporated the transient effects of the cost into the performance optimization problem.

4.1 Problem Statement

Virtualization-based server consolidation requires runtime resource reconfiguration to ensure adequate application isolation and performance, especially for multi-tier services that have dynamic, rapidly changing workloads and responsiveness requirements. While virtualization makes reconfiguration easy, indiscriminate use of adaptations such as VM cloning, VM migration, and CPU capacity enforcement has performance implications. However, there is little work that considers the impact of the reconfiguration actions themselves on application performance except in very limited contexts. For example, while [20] shows that live migration of VMs can be performed with a few milliseconds of downtime and minimal performance degradation, the results are limited only to Web servers. This can be very different for other commonly used types of servers.

4.1.1 Reconfiguration Overhead: Impact on Performance

Table 3 shows the impact of VM migration of servers from different J2EE-based tiers on the end-to-end mean response time of RUBiS computed over 3 minute intervals. Furthermore, because of interference due to shared resources, such migrations also impact the

Table 3: End-to-end response time (ms) during VM migration

Before Migration	Apache Migration	% Chg.	Tomcat Migration	% Chg.	MySQL Migration	% Chg.
102.92	141.62	37.60	315.83	206.89	320.93	211.83

performance of other applications whose VMs run on the same physical hosts (see Section 4.3.2). Cheaper actions such as CPU tuning can sometimes be used to achieve the same goals, however. These results indicate that the careful use of adaptations is critical to ensure that the benefits of runtime reconfiguration are not overshadowed by their costs.

This research work tackles the problem of optimizing resource allocation in consolidated server environments by proposing a runtime adaptation system that automatically reconfigures multi-tier applications running in virtualized data centers while considering adaptation costs and satisfying response-time-based SLAs even under rapidly changing workloads. The problem is challenging since the costs and benefits of reconfigurations are influenced not just by the software component targeted, but also by the reconfiguration action chosen, the application structure, its workload, the original configuration, and the application's SLAs.

We use automatic off-line experimentation presented in Chapter 2.4.3 to construct cost models that quantify the degradation in application performance due to reconfiguration actions. Using queuing models for predicting the benefit of a new configuration, we show how the cost models allow the analysis of cost-benefit tradeoffs to direct the online selection of reconfiguration actions. Then, we develop a best-first graph search algorithm based on the models to choose optimal sequences of actions. Finally, experimental results using RUBiS under different workloads derived from real Internet traces show that our cost-sensitive approach can significantly reduce SLA violations, and provide higher utility as compared to both static and dynamic-reconfiguration-based approaches that ignore adaptation costs.

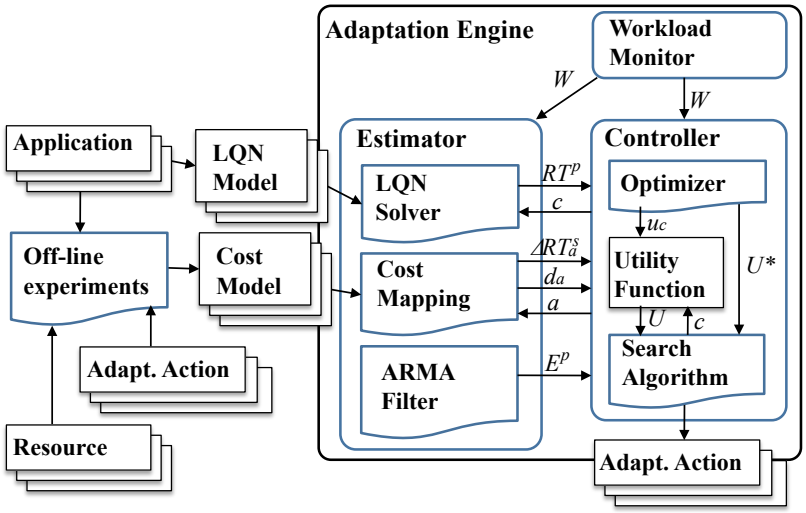


Figure 17: Cost-sensitive approach

4.2 Approach

4.2.1 Overview: Cost-Aware Optimization

To address the tradeoff between performance benefit and adaptation cost, the adaptation system estimates the cost and the potential benefit of each adaptation in terms of changes in the utility. Since the utility is a function of the mean end-to-end response time in this research work as shown in Equation 1 in Chapter 2.3, the *cost of adaptation* for a given adaptation depends on its duration and impact on the applications' response times. On the other hand, the *benefit of adaptation* depends on the change in applications' response times and how long the system remains in the new configuration.

The adaptation system manages the shared host pool by performing various *adaptation actions* such as CPU capacity tuning, VM live-migration, and component replication. As shown in Figure 17, it consists of a workload monitor, estimator, and controller. The workload monitor tracks the workload at the ingress of the system as a set of transaction request rates for each hosted application. The estimator consists of an LQN solver, a cost mapping, and an ARMA filter. The LQN solver uses layered queuing models [28] described in

Chapter 2.4.1 to estimate the mean response time RT^s for each application given a workload W and configuration c. The cost mapping uses cost models described in Chapter 2.4.3 to estimate the duration $d(a)$ and performance impact $\Delta RT^s(a)$ of a given adaptation a. Both types of models are constructed using the results of an off-line model parametrization phase. Finally, the ARMA (auto-regressive moving average) filter described in Chapter 2.4.4 provides a prediction of the *stability interval* E^p that denotes the duration for which the current workload will remain stable.

The controller invokes the estimator to obtain response time and cost estimates for an action's execution, which it uses to iteratively explore candidate actions. Using a search algorithm and the utility function, the controller chooses the set of actions that maximizes the overall utility. The search is guided by the upper bound on the utility U^* calculated using a previously-developed offline optimization algorithm described in Chapter 3.2.2 that provides the configuration that optimizes utility for a given workload without considering reconfiguration cost.

To balance the cost accrued over the duration of an adaptation with the benefits accrued between its completion and the next adaptation, the algorithm uses a parameter, called the *control window*, that indicates the time to the next adaptation (For a more specific formulation of the problem, see Chapter 2.3). Adaptations occur only because of controller invocations. If the controller is invoked periodically, the control window is set to the fixed inter-invocation interval. If the controller is invoked on demand when the workload changes, the control window is set to the stability interval prediction E^p provided by the ARMA filter. An adaptation is only chosen if it increases utility by the end of the control window. Therefore, a short control window produces a conservative controller that will typically only choose cheap adaptation actions, while a longer control window allows the controller to choose more expensive adaptations.

A two-level controller is implemented to achieve a balance between rapid but cheap response to short term fluctuations and more disruptive responses to long term workload

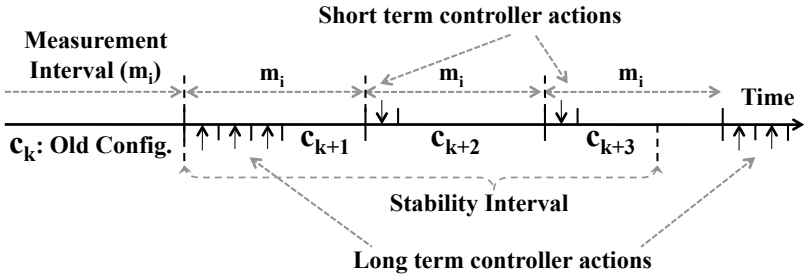


Figure 18: Control timeline

changes (Figure 18). The *short term* controller is invoked periodically once every measurement interval, while the *long term* controller is executed on-demand when the workload has changed more than a specified threshold since the last long term controller invocation. To avoid multiple controller executions in parallel, the timer tracking the short term controller's execution is suspended while the long term controller is active.

4.2.2 Search Algorithm

The goal of the search algorithm is to find a configuration (and the corresponding adaptation actions) for which the utility U is maximized. Configurations must satisfy the following allocation constraints: (a) for each application, only one replica from each tier can be assigned to a host, (b) the sum of CPU allocations on a host can be at most 1, and (c) the number of VMs per host is restricted to fit the available memory on the host.

Starting from a current configuration, a new configuration at each step is built by applying exactly one adaptation action as shown in Figure 19. The vertices represent system configurations, and the edges represent adaptation actions. We frame the problem as a shortest path problem that minimizes the negative of the utility, i.e., maximizes the actual utility. Therefore, each edge has a weight corresponding to the negative of the utility obtained while the action is being executed. If multiple action sequences lead to the same configuration, the vertices are combined. Configurations can be either intermediate or candidate configurations as represented by the white and gray circles in the figure, respectively.

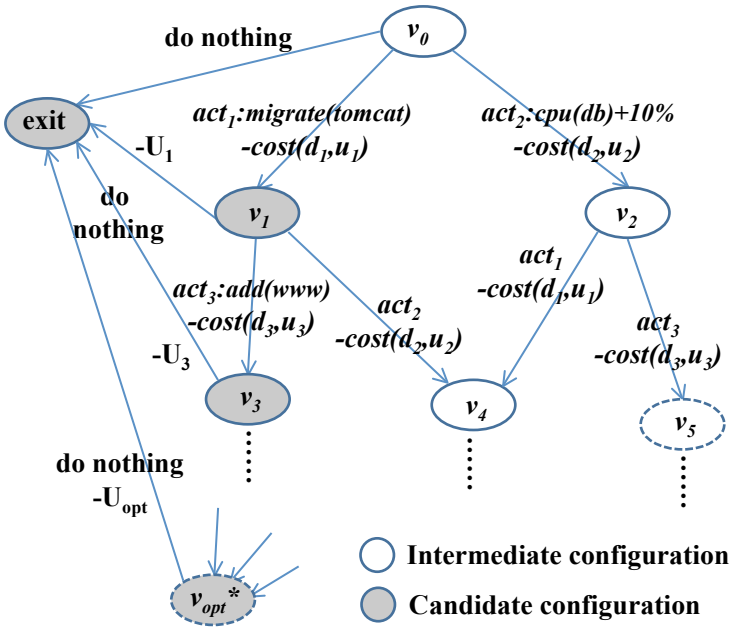


Figure 19: Adaptation action search graph

A candidate configuration satisfies the allocation constraints, while an intermediate configuration does not, e.g., it may assign more CPU capacity to VMs than is available, requiring a subsequent "Reduce CPU" action to yield a candidate configuration. Neither type of configuration is allowed to have individual replicas with CPU capacity greater than one.

Only candidate configurations have a `do nothing` action that leads the goal state, labeled as `exit` in the figure. The weight for the `do nothing` action in a configuration c is the negative of the revenue obtained by staying in c until the end of the prediction interval (i.e. $-U_c$), assuming that the best known path is used to get to c. Then, the shortest path starting from the initial configuration to the `exit` state computes the best U, and represents the adaptation actions needed to achieve optimal revenue. Intermediate configurations do not have `do nothing` actions, and thus their utility is not defined.

Although the problem reduces to a weighted shortest path problem, it is not possible to fully explore the extremely large configuration space. To tackle this challenge without sacrificing optimality, we adopt an A* best-first graph search approach as described in [64]. The approach requires a "cost-to-go" heuristic to be associated with each vertex of the

graph. The cost-to-go heuristic estimates the shortest distance from the vertex to the goal state (in our case, the `exit` vertex). It then explores the vertex for which the estimated cost to get to the goal (i.e., the sum of the cost to get to the vertex and the cost-to-go) is the lowest. In order for the result to be optimal, the A* algorithm requires the heuristic to be "permissible" in that it underestimates the cost-to-go.

As the cost-to-go heuristic, we use the utility u^* of the optimal configuration c^* that is produced by our previous work in Chapter 3 using bin-packing and gradient-search techniques. This utility value represents the highest rate at which utility can be generated for the given workload and hardware resources. However, it does not take into account any costs that might be involved to change to that configuration, and thus overestimates the utility that can practically be obtained in any given situation. Therefore, the utility U calculated by using u^* instead of accrued utilities during the rest of control window is guaranteed to overestimate the true reward-to-go (i.e., underestimate cost-to-go), and thus forms a permissible heuristic.

Input: c_i: current config., W_i: predicted workload, CW: control window length
Output: $\mathcal{A}^i_{opt}$ - the optimized adaptation action sequence
$(c^*, u^*) \leftarrow$`UtilityUpperBound` (W_i)
if $c^* = c_i$ **then return** $[a_{null}]$ (do nothing)
$v^0.(a_{opt}, c, U(a), U, D) \leftarrow (\phi, c_i, 0, u^*, 0)$; $\mathcal{V} \leftarrow \{v^0\}$
while *forever* **do**
  $v \leftarrow \text{argmax}_{v' \in \mathcal{V}} v'.U$
  if $v.a_{opt}[\text{last}] = a_{null}$ **then return** $v.a_{opt}$
  foreach $a \in \mathcal{A} \cup a_{null}$ **do**
    $v^n \leftarrow v$, $v^n.a_{opt} \leftarrow v.a_{opt} + a$
    if $a = a_{null}$ **then**
      $u_c \leftarrow$ `LQNS` $(W_i, v^n.c)$; $v^n.U \leftarrow (CW - v^n.D) \cdot u_c + v^n.U(a)$
    else
      $v^n.c \leftarrow$ `NewConfig` $(v^n.c, a)$; $(d(a), u(a)) \leftarrow$`Cost` $(v^n.c, a, W_i)$
      $v^n.U(a) \leftarrow v^n.U(a) + d(a) \cdot u(a)$; $v^n.D \leftarrow v^n.D + d(a)$;
      $v^n.U \leftarrow (CW - v^n.D) \cdot u^* + v^n.U(a)$
  if $\exists v' \in \mathcal{V}$ *s.t.* $v'.c = v^n.c$ **then**
    if $v'.U > v^n.U$ **then** $v' \leftarrow v^n$
  else
    $\mathcal{V} \leftarrow \mathcal{V} \cup v^n$

Algorithm 2: Optimal adaptation search

The resulting search algorithm is shown in Algorithm 3. After using the UtilityUpperBound function to compute the cost-to-go heuristic u^* for the initial configuration v^0, it begins the search. In each iteration, the open vertex with the highest value of U is explored further. New open vertices are created by applying each allowed adaptation action to the current vertex and updating $v.a_{opt}$, the optimal list of actions to get to v. When applying the `do nothing` action, the algorithm invokes the LQNS solver to estimate the response times of the current configuration and computes the utility. Otherwise, it invokes NewConfig to produce a new configuration and uses the cost model to compute both the adaptation cost $U(a)$ and the overall utility U as explained above. The algorithm terminates when $a.null$, i.e., "do nothing", is the action chosen.

4.2.3 Reducing the Search Space

The running time of the algorithm depends on the number of configurations explored by the search. The algorithm avoids lengthy sequences of expensive actions due to the optimal utility bound. However, to prevent it from getting stuck exploring long sequences of cheap actions such as CPU allocation changes, we have implemented several techniques to significantly reduce the number of states generated without affecting the quality of the adaptations produced. The first is *depth limiting* (DL), which limits the search of paths to those of no more than n adaptation actions and effectively makes the search space finite. In our experiments, we chose $n = 7$ as the largest value that ensured that the controller always produced a decision within 30 seconds. The second is *partial order reduction* (PO), which addresses the issue that CPU tuning actions can interleave in many ways to produce the same results, but require different intermediate states, e.g., WS+10%, WS+10%, DB-10% and DB-10%, WS+10%, WS+10%. To prevent multiple inter-leavings without affecting the actual candidate configurations, we consider all CPU increases and decreases in a strict canonical order of components. The final technique is *action elimination* (AE), which eliminates known poor action choices, for example, disabling add replica actions

Table 4: State space reduction

Technique	States	Time (sec)
Naive	83497	3180
DL	19387	1420
DL+PO	599	210
DL+PO+AE	62	18

when the workload for an application has diminished.

Table 4 shows the magnitude of reductions that are achievable with these techniques using an experiment in which 10 VMs across two applications were being optimized. Adding more replicas to an application does not affect the size of the state-space. However, adding more applications does. While these results indicate that the search algorithm can be made fast enough to be used in an on-line manner while still retaining a high quality of adaptation for deployments of small to moderate size, scalability is potentially a problem for large deployments.

4.3 Evaluation Results

The experimental results are divided into three parts. In the first part, we describe the testbed and workloads used, and then present the measurements used in the adaptation cost models. In the second part, we evaluate the accuracy of the individual controller components in the testbed: the LQNS performance models, the cost models, and the ARMA-based workload stability predictor. Finally, in the third part, we evaluate our approach holistically in terms of the quality of the adaptation decisions the controller produces and their impact on application SLAs.

4.3.1 Experimental Setup

Our target system is a three-tier version of the RUBiS application. The application consists of Apache web servers, Tomcat application servers, and MySQL database servers running

on a Linux-2.6 guest OS using the Xen 3.2 virtualization platform. The hosts are commodity Pentium-4 1.8GHz machines with 1GB of memory running on a single 100Mbps Ethernet segment. Each VM is allocated 256MB of memory, with a limit of up to 3 VMs per host. The Xen Domain-0 hypervisor is allocated the remaining 256MB. The total CPU capacity of all VMs on a host is capped to 80% to ensure enough resources for the hypervisor even under loaded conditions. Our test-bed is built by a small cluster of hosting machines as illustrated in Figure 4 in Chapter 2.2. Five machines are used to host our test applications, while two are used as client emulators to generate workloads. One machine is dedicated to hosting dormant VMs used in server replication, and another one is used as a storage server for VM disk images. Finally, we run the adaptation engine on a separate machine with 4 Intel Xeon 3.00 GHz processors and 4 GB RAM. For MySQL replication, all tables are copied and synchronized between replicas. The Tomcat servers are configured to send queries to the MySQL replicas in a round-robin manner. We deploy two applications RUBiS-1 and RUBiS-2 in a default configuration that evenly allocates resources among all components except for the database servers, which are allocated an additional 20% CPU to avoid bottlenecks. The target response time (84 ms in these experiments) was derived experimentally as the mean response time across all transactions of a single RUBiS application running in isolation in the initial configuration driven by a constant workload equal to half of the design workload range of 5 to 80 requests/sec.

During experiments, we drive the target applications using two workloads, a time-of-day workload and a flash crowd workload. The time-of-day workload was generated based on the Web traces from the 1998 World Cup site [5] and the traffic traces of an HP customer's Internet Web server system [23]. We have chosen a typical day's traffic from each of these traces and then scaled them to the range of request rates that our experimental setup can handle. Specifically, we scaled both the World Cup requests rates of 150 to 1200 requests/sec and the HP traffic of 2 to 4.5 requests/sec to a range of 5 to 80 requests/sec. Since our workload is controlled by adjusting the number of simulated clients, we created

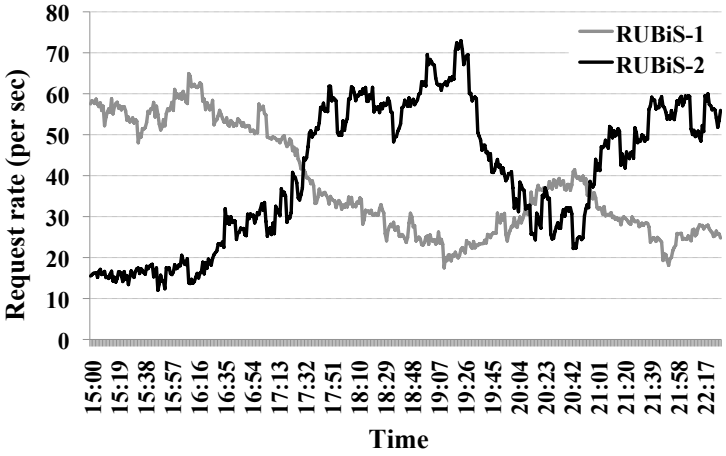


Figure 20: Time-of-day workload

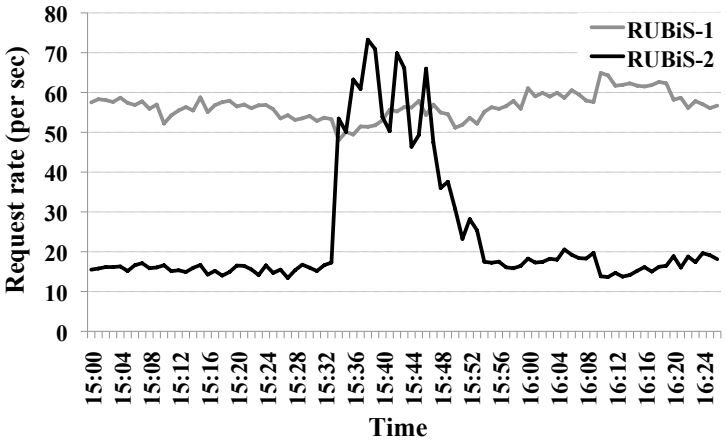
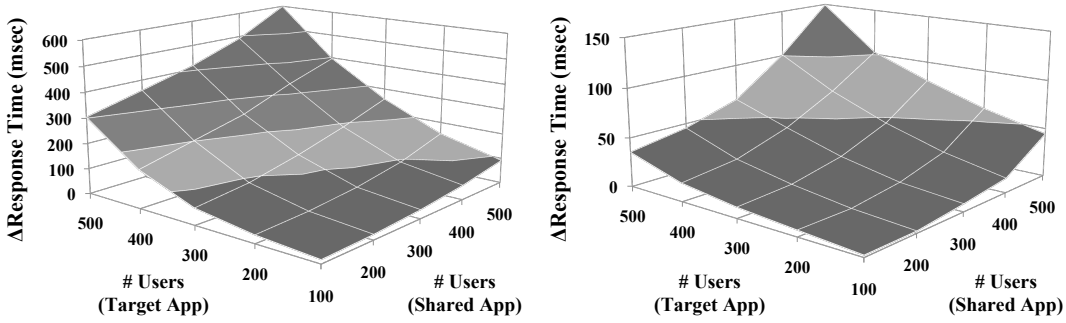


Figure 21: Flash crowd workload

a mapping from the desired request rates to the number of simulated RUBiS clients. Figure 20 shows these scaled workloads for the two RUBiS applications from 15:00 to 22:30, where RUBiS-1 uses the scaled World Cup workload profile and RUBiS-2 uses the scaled HP workload profile. The flash crowd workload shown in Figure 21 uses the first 90 minutes of the time-of-day workloads, but has an additional load of over 50 requests per second added to RUBiS-2 around 15:30 for a short interval.



(a) Target App ΔRT (b) Shared App ΔRT

Figure 22: Delta response times of MySQL live-migration

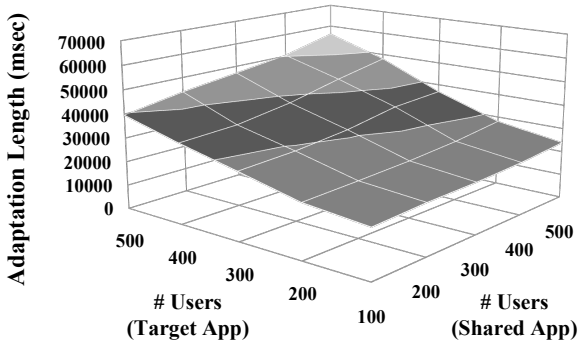


Figure 23: Adaptation duration of MySQL live-migration

4.3.2 Adaptation Costs: Performance Degradation

To measure adaptation costs, we deployed both applications and used the methodology described in Chapter 2.4.3. One application was the "target application" for the action, while the other was the "shared application" that was co-located with the target application, but was not reconfigured. We measured the adaptation length $d(a)$ and response time impact $\Delta RT^s(a)$ for all adaptation actions and combinations of workloads ranging from 100 to 500 users for both the target and shared application.

For example, Figures 22 and 23 show $\Delta RT^s(a)$ and $d(a)$ for the target application when subjected to actions affecting the MySQL server and when the workload for both applications is increased equally. As is seen, ΔRT for adding and removing MySQL replicas increases as workloads increase, but the adaptation durations are not greatly affected. The

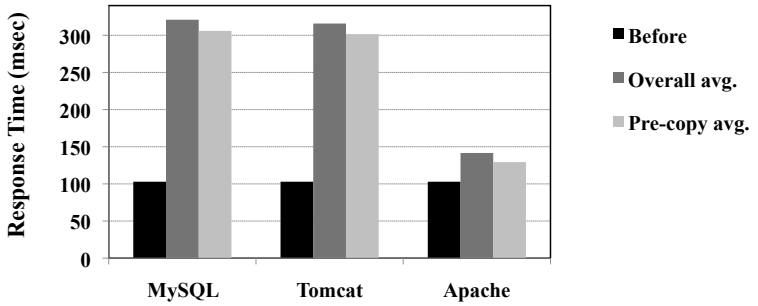


Figure 24: Live migration costs of various servers

costs of CPU reallocation are very small in terms of both ΔRT and $d(a)$.

The most interesting results were those for live migration, which has been proposed in the literature as a cheap technique for VM adaptation (e.g., [20]). However, we see that live-migration can have a significant impact on a multi-tier application's end-to-end responsiveness both in magnitude and in duration. For each of the three server types, Figure 24 shows the mean end-to-end response time for RUBiS measured before migration of that server, over the entire migration duration, and during the "pre-copy" phase of migration. This figure shows that although live-migration is relatively cheap for the Apache server, it is very expensive for both the Tomcat and MySQL servers. Moreover, most of this overhead incurs during the pre-copy phase. During this phase, dirty pages are iteratively copied to the target machine at a slow pace while the VM is running. In the subsequent stop-and-copy phase, the VM is stopped and the remaining few dirty pages are copied rapidly. Claims that VM migration is "cheap" often focus on the short (we measured it to be as low as 60msec) stop-and-copy phase when the VM is unavailable. However, it is the much longer pre-copy phase with times averaging 35 sec for Apache, 40 sec for MySQL, and 55 sec for the Tomcat server that contributes the most to end-to-end performance costs.

Migration also affects the response time of other VMs running on the same host. Figures 22(a) and 22(b) show the ΔRT for the target and shared applications, respectively during MySQL migration. While increases in the shared application's number of users

Table 5: Variance of adaptation costs for MySQL migration

Workload	Action Length	RUBiS-1 ΔRT	RUBiS-2 ΔRT
100:500	2.34%	2.95%	14.52%
300:500	7.45%	10.53%	17.14%
500:500	8.14%	6.79%	101.80%

(i.e., workload) impact the target application's response time, the target application migration has an even more significant impact on the shared application, especially at high workloads. Figure 23 shows how the adaptation duration increases with the target workload due to an increase in the working set memory size.

In Table 5, we also show the standard deviations for these costs as percentages of the mean and calculated across all the random configurations used for measurement. The variances are quite low indicating that exact knowledge of the configuration does not significantly impact migration cost, and validating our cost model approximations. The only outlier we saw was for the response time of RUBiS-2 when two MySQL servers were co-located under high load.

4.3.3 Model Prediction Accuracy

We evaluate the accuracy of the LQN models and the cost models in a single experiment by using the first 220 minutes from the time-of-day workloads. Specifically, at each controller execution point and for each application, we recorded the response time predicted by the models (RT^s) for the next control interval and then compared it against the actual measured response time over the same time period. This comparison includes both the predictions of adaptation cost and performance. Figure 25 shows the results for each application. Despite the simplifications made in our cost models, the average estimation error is quite good at around 15%, with the predictions being more conservative than reality.

Second, we evaluated the accuracy of our stability interval estimation. To do this, the

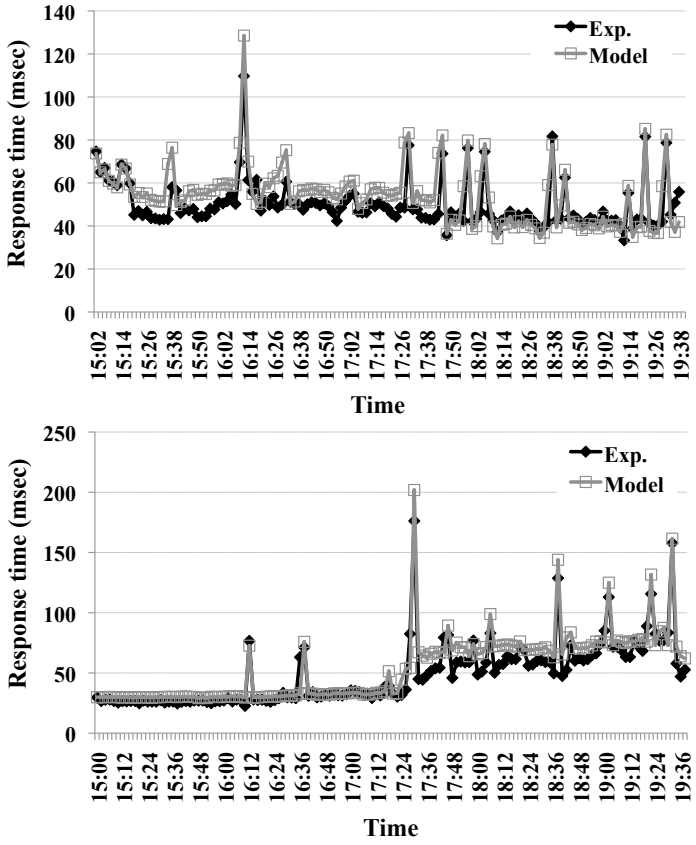


Figure 25: Prediction accuracy for both applications under time-of-day workload

ARMA filter is first trained using 30 minutes of the respective workloads. As shown in Figure 26, the filter is executed 68 times during the time-of-day experiment and provides effective estimates. The absolute prediction error against the measured interval length is around 15% for the time-of-day workloads. Meanwhile, the flash crowd workload causes an increase in the estimation error of the ARMA filter due to the short and high unexpected bursts. The results are presented in Figure 27. The error reaches approximately 23% because the filter over-estimates the length until the 5^{th} stability interval when the flash crowd appears. However, the estimation quickly converges on the lower length and matches the monitored length of the stability interval until the 14^{th} interval, when the flash crowd goes away and the filter starts to under-estimate the length. Even under such relatively high prediction errors, we show below that our cost-sensitive strategy works well.

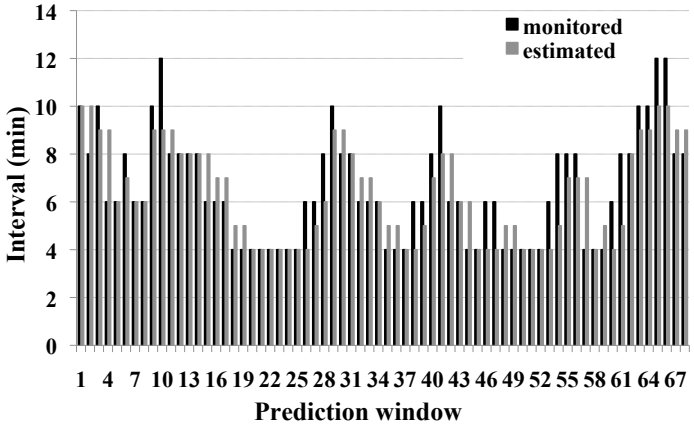


Figure 26: Stability interval prediction errors in time-of-day workload

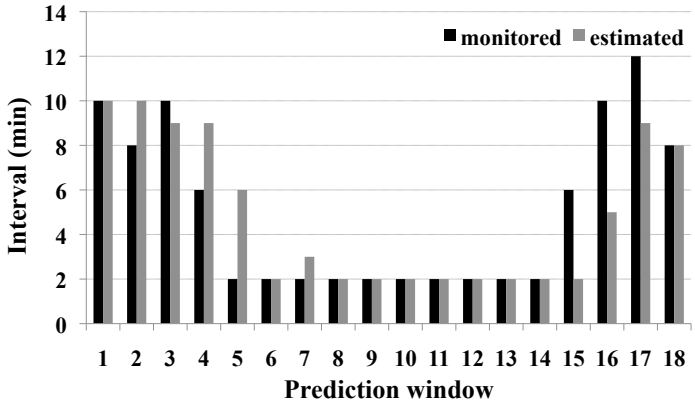


Figure 27: Stability interval prediction errors in flash crowd workload

4.3.4 Controller Evaluation

We evaluate our Cost-Sensitive (CS) strategy under both time-of-day workload and flash crowd scenarios by comparing its response time and utility against the following strategies:

- Cost Oblivious(CO): reconfigures the system to the optimal configuration whenever the workload changes, and uses the optimal configurations generated using our previous work (see Chapter 3).

- 1 Hour: reconfigures the system to the optimal configuration periodically at the rate of once per hour; this strategy reflects the common policy of using large consolidation windows to minimize adaptation costs.

- No Adaptation(NA): maintains the default configuration throughout the experiment.
- Oracle: provides an upper bound for utility by optimizing based on perfect knowledge of future workload and by ignoring all adaptation costs.

We use the current measured workload at the controller execution point to be the predicted workload for the next control window for the CS and CO strategies. The measurement interval is set to 2 minutes to ensure quick reaction in response to workload changes. The workload monitor gets the workload for each measurement interval by parsing the Apache log file. Finally, we choose a narrow workload band b of 4 req/sec to ensure that even small workload changes will cause the controller to consider taking action.

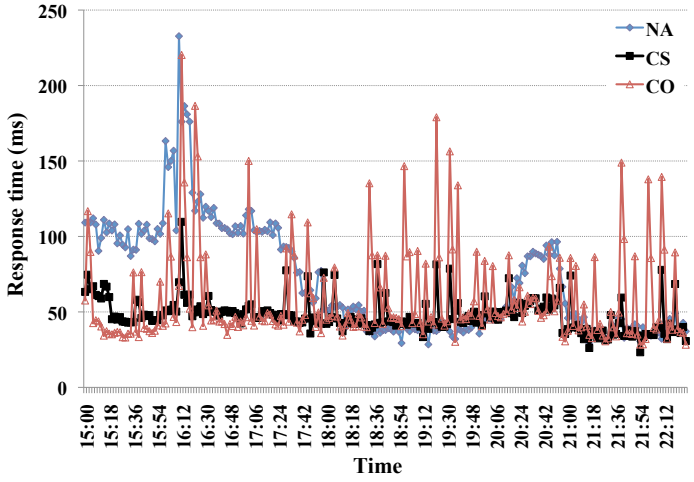


Figure 28: Response time comparison in time-of-day workload

End-to-End Response Time. First, we compare the mean end-to-end response time for all the strategies as measured at each measurement period. The results for the RUBiS-1 application are shown for the CS, CO, and NA strategies in Figures 28 and 29; the oracle and 1-Hour plots are omitted for legibility. Figure 28 shows the results for the time-of-day workload. Predictably, the NA strategy is very sensitive to workload changes and shows large spikes once the workload intensity reaches the peak in both applications. For the CO and CS strategies, a series of spikes corresponds to when the adaptation engine triggers

adaptations. The CS strategy has relatively short spikes and then the response time stabilizes. Meanwhile, the CO strategy has more and larger spikes than the CS strategy. This is because the CO strategy uses more adaptation actions, including relatively expensive ones such as live-migration of MySQL and Tomcat and MySQL replication, while the CS strategy uses fewer and cheaper actions, especially when the estimated stability interval is short. Although the response time of the CO strategy is usually better than the CS strategy after each adaptation has completed, the overall average response time of CS is 47.99 ms, which is much closer to the oracle's result of 40.91ms than the CO, 1-Hour, and NA values, which are 58.06 ms, 57.41 ms, and 71.18 ms respectively.

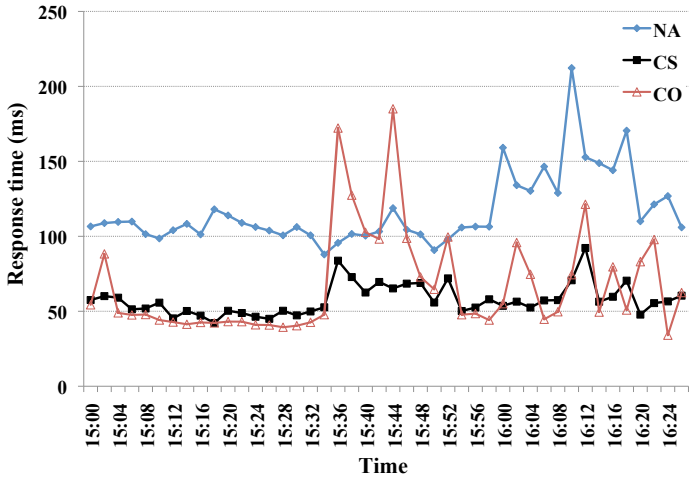


Figure 29: Response time comparison in flash crowd workload

Similarly, for the flash crowd scenario, although the ARMA filter over- and under-estimates several stability intervals, the CS strategy's mean response time of 57.68 ms compares favorably with the CO, 1-Hour, and NA values of 67.56 ms, 70.42 ms, and 116.35 ms, respectively, and is closer to the oracle result of 40.14ms as shown in Figure 29. Not surprisingly, the difference between CS and oracle is larger for the flash crowd workload than the time-of-day one because the ARMA filter is wrong more often in its stability interval predictions. Also, the 1-Hour strategy does more poorly in the flash crowd case because it is unable to respond to the sudden workload spike in time. The results for

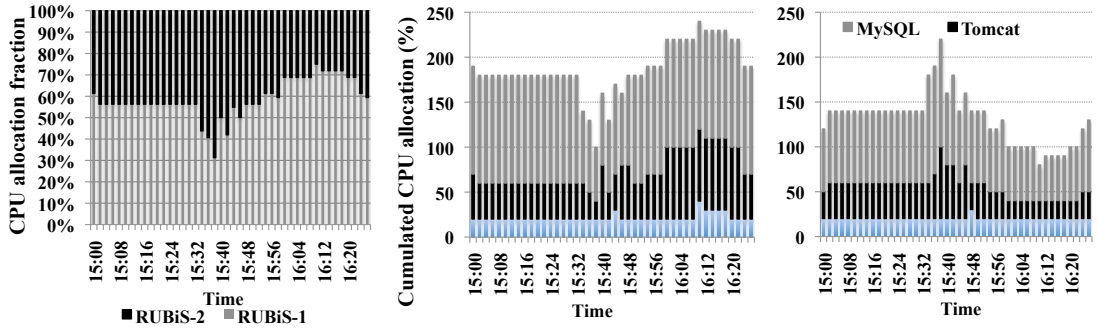


Figure 30: CPU allocations of oracle

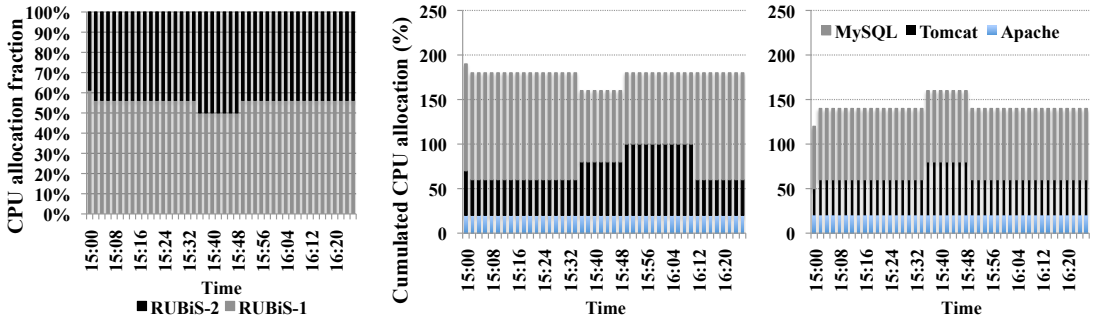


Figure 31: CPU allocations of the CS strategy

RUBiS-2 were similar. Thus, the CS controller is able to outperform the CO strategy over the long run by trading-off short-term optimality for long-term gain.

To illustrate how the different strategies affect adaptation behaviors in the flash crowd scenario, we look at resource allocation for the CS strategy versus oracle. Figures 30 and 31 show the CPU allocation between applications (the leftmost plot in both figures) as well as among their components over time (the middle plot representing RUBiS-1 and the

Table 6: Total number of actions triggered

Action	CS	CO
CPU Increase/Decrease	14	36
Add (MySQL replica)	1	4
Remove (MySQL replica)	1	4
Migrate (Apache replica)	4	10
Migrate (Tomcat replica)	4	10
Migrate (MySQL replica)	0	2

rightmost plot representing RUBiS-2 in both figures). As shown in these figures, oracle moves more CPU resources between the two applications and also among the components in each application using more expensive actions than the CS strategy. In particular, when the load to RUBiS-2 suddenly increases around 15:30, oracle removes a MySQL replica of RUBiS-1 and adds a MySQL replica to RUBiS-2. Note that, if a component has more than 100% CPU allocation (e.g., MySQL in middle and right most graphs of Figure 30), it is replicated and uses more than one physical machine. Meanwhile, the CS strategy also removes the MySQL replica of RUBiS-1, but after that it only tunes the CPU allocation of Tomcat servers, which are much cheaper actions than adding a replica to RUBiS-2. The middle graph of Figure 31 shows that the CPU allocation of MySQL in RUBiS-1 is reduced from 120% to 80% at 15:36, but that the CPU allocation of MySQL in RUBiS-2 (see the rightmost graph) is not changed. Table 6 summarizes the actions used to adapt configurations by the CS and CO strategies for the flash crowd scenario. Although the CS strategy uses fewer actions than the CO strategy, its average response time is lower than that of the CO strategy.

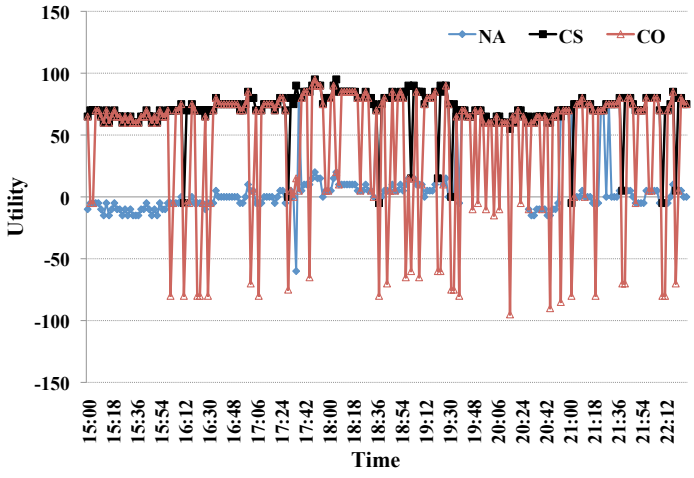


Figure 32: Utility comparison in time-of-day workload

Utility. Using the monitored request rates and response times, we compute the utility of each strategy at every measurement interval to show the impact of adaptation actions on

the overall utility. For the time-of-day workload, Figure 32 shows that both the CO and CS strategies have spikes when adaptation actions are triggered. However, the CO strategy has more and much deeper spikes than the CS strategy including some that lead to negative utility by violating SLAs of both applications. Meanwhile, the CS strategy chooses actions that do not violate SLAs.

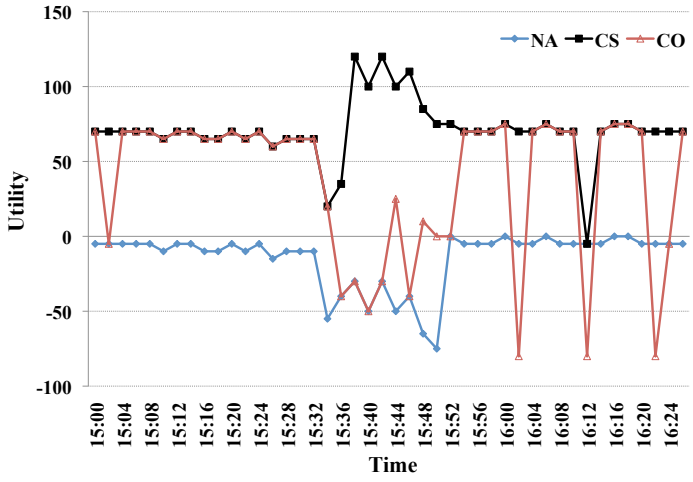


Figure 33: Utility comparison in flash crowd workload

The utility for the flash crowd scenario in Figure 33 similarly shows that the CS strategy has a couple of spikes corresponding to the onset and exit of the flash crowd. However, these spikes are less severe than those of the CO strategy. The CS strategy violates the SLA of RUBiS-1 only in the measurement periods where it removes or adds a MySQL replica of RUBiS-1 (when the flash crowd starts and then after it disappears), while the CO strategy violates SLAs of both applications in many periods.

We also computed the total utility accumulated over the entire experiment duration. The values for all the different strategies and workloads are shown in Table 7. Because the absolute value of the utility can differ greatly depending on the exact reward, penalty, and response time threshold values used in the SLA, it is more important to note the relative ordering between the different approaches. As can be seen, the CS strategy performs the

Table 7: Cumulative utility for all strategies

Workload	Oracle	CS	1 Hour	CO	NA
Time of day	16535	15785	10645	9280	2285
Flash Crowd	3345	3120	2035	1620	-630

best and has a utility very close to the oracle for both workloads. The NA strategy predictably performs the worst. While neither the CO nor the 1-Hour strategy are competitive with CS, it is interesting to note that CO performs worse than 1-Hour. This is because CO is so aggressive in choosing optimal configurations that it incurs too much adaptation cost compared to 1-Hour, which limits adaptations to once every hour. The higher frequency of response time spikes for the CO and NA approaches indicates that this ordering is not likely to change even if a different utility function is used. These results demonstrate the value of taking workload stability and costs into account when dynamic adaptations are made.

4.4 Work Related to Cost-Sensitive Adaptation

The primary contributions of this research work are (a) a model for comparing on a uniform footing dramatically different types of adaptation actions with varying cost and application performance impacts (e.g., CPU tuning vs. VM migration), and (b) considering workload stability to produce adaptations that are not necessarily optimal in the short term, but produce better results over the long run when workload variations are taken into account. We are not aware of any other work that addresses these issues, especially in the context of multi-tier systems with response time SLAs.

Several papers address the problem of dynamic resource provisioning [4, 15, 82, 8, 86, 88, 74]. The authors in [74] even use queuing models to make decisions that preserve response time SLAs in multi-tier applications. However, none of these papers consider the performance impact of the adaptations themselves in their decision making process. The approach proposed in [69] learns the relationships between application response time, workload, and adaptation actions using reinforcement learning. It is implicitly able to learn

adaptation costs as a side-benefit. However, it cannot handle never-before seen configurations or workloads, and must spend considerable time relearning its policies in case of even workload changes.

Recently, some efforts including [48, 83, 83, 75, 31] address adaptation costs. Only one adaptation action, VM migration, is considered in [48], [83], and [32]. These papers propose controllers based on online vector-packing, utilization to migration cost ratios, and genetic algorithms, respectively, to redeploy components whose resource utilization causes them to fit poorly on their current hosts while minimizing the number or cost of migrations. Migrations are constrained by resource capacity considerations, but once completed, they are assumed not to impact the subsequent performance of the application. Therefore, the approaches cannot be easily extended to incorporate additional action types since they possess no mechanisms to compare different performance levels that could result from actions such as CPU tuning or component addition. pMapper focuses on optimizing power given fixed resource utilization targets produced by an external performance manager [75]. It relies solely on VM migration, and propose a variant of bin-packing that can minimize the migration costs while discarding migrations that have no net benefit. It also does not provide any way to compare the performance of different types of actions that achieve similar goals. Finally, [31] examines an integer linear program formulation in a grid job scheduler setting to dynamically produce adaptation actions of two types - VM migration and application reconfiguration - to which users can assign different costs. However, there is again no mechanism to compare the different performance benefits of the different actions, and the user must resort to providing a manual weight to prioritize each type of action.

In summary, the above "cost aware" approaches only minimize adaptation costs while maintaining fixed resource usage levels. They do not provide a true cost-performance tradeoff that compares different levels of performance resulting from different kinds of actions. Furthermore, none of the techniques consider the limited lifetime that reconfiguration is

likely to have under rapidly changing workloads and adjusts its decisions to limit adaptation costs accordingly. In that sense, they are more comparable to our "cost oblivious" policy which reconfigures the system whenever it finds a better configuration for the current workload, irrespective of future trends.

The only work we are aware of that explicitly considers future workload variations by using a limited lookahead controller (LLC) is presented in [51]. The algorithm balances application performance with energy consumption by switching physical hosts on and off. However, it only deals with a single type of coarse grain adaptation action, and requires accurate workload predictions over multiple windows into the future, something that is hard to get right. In contrast, our approach does not require any workload predictions, but can benefit from much simpler to obtain estimates of stability windows if they are available. Moreover, it is not clear whether it is practical to extend the LLC approach to allow multiple types of actions with a range of granularities.

Energy saving is considered an explicit optimization goal in [51] and [75] and is realized by shutting down machines when possible. Our approach does not factor in the cost of energy and therefore does not consider power cycling actions. CPU power states are virtualized in [56] to produce "soft power states" exported by an hypervisor to its VMs. In this approach, each VM implements its own power management policy through the soft-states, and the management framework arbitrates requests from multiple VMs to either perform frequency scaling, or VM capacity scaling along with consolidation. It leaves policy decisions, i.e., (a) how performance goals and workload are mapped to resource targets, and (b) when and which VMs are consolidated to which physical hosts, to the application to decide. Our goal is to automatically produce such policies.

4.5 Summary

In this research work, we have shown that runtime reconfiguration actions such as virtual machine replication and migration can impose significant performance costs in multi-tier

applications running in virtualized data center environments. To address these costs while still retaining the benefits afforded by such reconfigurations, we developed an adaptation system for generating cost-sensitive adaptation actions using a combination of predictive models and graph search techniques. Through extensive experimental evaluation using real workload traces from Internet applications, we have showed that by making smart decisions on when and how to act, the approach can significantly enhance the satisfaction of response time SLAs compared to approaches that do not take adaptation costs into account.

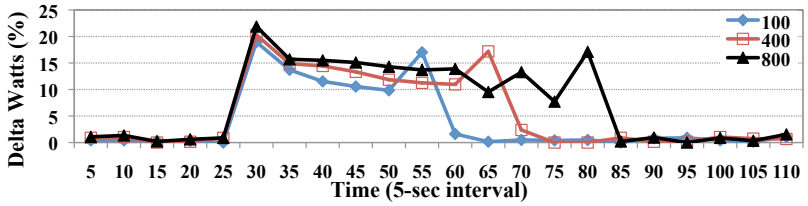
CHAPTER V

MULTI-DIMENSIONAL OPTIMIZATION

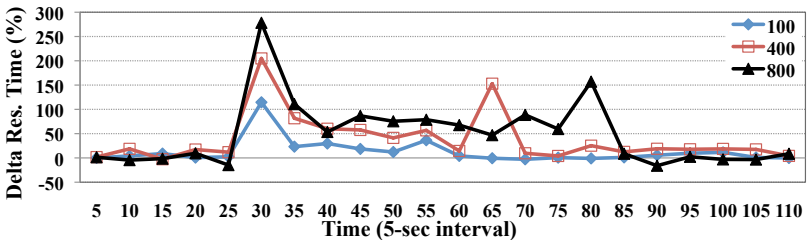
In our prior research work presented in Chapters 3 and 4, we have addressed the optimization problem only related to the application performance. The research work described in this chapter integrates the power consumption into the optimization formulation since the power consumption has become equally important for cloud computing as is performance. This chapter presents, first, the tradeoff between performance and power consumption, and shows experimental results that illustrate various adaptation overheads in the context of not only the end-to-end response time but also additional power consumption. Then, the multi-dimensional optimization approach, referred to as $Mistral$, is discussed in detail. Finally, we show the evaluation result of our approach by comparing it with other optimization approaches.

5.1 Problem Statement

Power consumption has recently become one of the top concerns in data center and cloud computing environments. With the rapid growth of data center deployments at scale, the total power consumption has doubled from 2000 to 2005. This significant power consumption rate of data centers has directly been caused by deploying the large number of inexpensive hosting servers [2, 1]. The rated power consumption of hosting servers has increased by 10 times over the past ten years. A recent Internet Data Center (IDC) report has estimated the worldwide spending on enterprise power to be more than $30 billion and likely to even surpass spending on new server hardware. Therefore, the performance is no longer only criterion for optimization and management in such environments. In response to the increased importance of the power savings, cloud infrastructure providers need to factor in the power consumption when managing deployed resources.



(a) Power consumption



(b) Response time

Figure 34: Costs of a single VM live-migration

5.1.1 Power-Performance Tradeoff

Server consolidation based on virtualization technology has become a key ingredient for improving power efficiency and resource utilization in cloud computing environments. Through adaptation actions such as VM migration and resource capacity control, cloud infrastructure providers not only accommodate demand spikes by temporarily taking resources away from underutilized applications, but also save power by consolidating servers into fewer number of resources. However, the indiscriminate use of server consolidation can adversely affect the application performance because of the inherent tradeoff between power consumption and performance. Thus, it must be used very carefully in such environments that provide a wide array of performance-sensitive services such as online portals and enterprise applications.

5.1.2 Reconfiguration Overhead: Impact on Power and Performance

In addition to the power-performance tradeoff, infrastructure providers must also consider the tradeoff between the cost of an adaptation and its benefit, since workload varies dynamically, and runtime consolidation actions such as migration are not free. For example,

Figure 34 shows the increase in power consumption and end-to-end response time of a 3-tier application as a function of time during the live migration of a single of the application's Xen-based VM. The VM migration is initiated at the 25sec mark in the figure. The measurements, shown for three different workload intensities of 100, 400, and 800 concurrent user sessions, indicate that the impact is not only significant, but that it depends on the workload and is incurred over a substantial period of time.

Coupled with a changing workload, adaptation costs can lead to complete rethinking of the best strategy for power savings. For example, when the workload is rapidly changing, it may be better to suffer a slight performance degradation rather than trigger an expensive migration whose costs may never be recouped before another adaptation is needed. Or, a cheap but modest change such as the redistribution of resources amongst VMs may be a more effective than powering up a new host. Additionally, the power cost and decision delay incurred by the system making the adaptation decision must also be considered. Often it may be better to make a suboptimal decision quickly rather than invest time and energy searching for savings that are not enough to recoup the investment. Therefore, infrastructure providers must account for balancing steady state performance and power with the dynamic adaptation costs under changing workloads.

5.2 Approach

In this dissertation, a multi-dimensional optimization approach is developed to balance power consumption, application performance, and transient power/performance costs due to adaptation actions and decision making in a single unified framework. By doing so, it can dynamically choose optimal adaptation actions from a variety of actions with differing effects in a multiple multi-tier application, dynamic workload environment. The cost-sensitive adaptation approach described in Chapter 4 is extended to incorporate the cost of power in both steady state and during adaptations, and enable significant power savings by migrating applications away from idle resources and shutting them down only

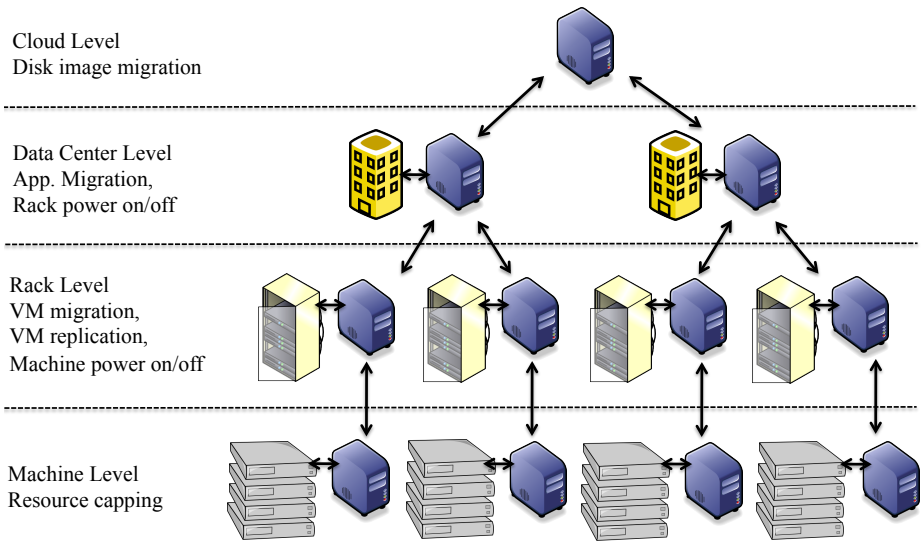


Figure 35: Architecture of 4-level control hierarchy

when appropriate. Also, this new approach provides an ability to factor the optimization system's power consumption and decision delays into its decision making. Finally, the adaptation system is constructed as a scalable, multi-level hierarchical system that can deal with a large number of applications and hosts, and also with adaptation actions at multiple time-scales ranging from a few milliseconds to tens of minutes.

5.2.1 System Architecture

We assume that a large-scale cloud computing environment can be built as a resource hierarchy (see Chapter 2.2). The optimization system is then deployed in the resource hierarchy as the form of a multi-level hierarchical control scheme with multiple instances of controllers. Each controller manages different subsets of hosts and applications, and operates at different time-scales. Figure 35 illustrates an example control hierarchy and the types of available adaptation actions generated at each level. The higher level controller can generate all types of actions that can be generated at the lower level controllers. For example, a rack level controller manages a small number of machines and applications by triggering VM migration and resource capping that can be triggered by a machine level controller. At the data center level, a controller manages all machines hosted in multiple

racks by triggering application migration and VM migration.

To understand how the controllers interact, consider that an adaptation action is only chosen by a controller if it is anticipated to increase utility over the next control window CW_i time units. From Equation 4 in Chapter 2, it can be seen that as the stability interval becomes longer, adaptation is less frequent, but the benefits of adaptation can accrue for longer periods. Thus, longer stability intervals make increasingly disruptive actions with potentially more significant benefits (e.g., application migrations, power cycling) worthwhile, while short stability intervals may rule out all but the quickest actions (e.g., CPU capacity tuning). Stability intervals can be made longer by making the workload bands wider (i.e., allowing a larger change in workload before adaptation is needed). Therefore, the lower level controllers are configured with very narrow workload bands and coupled with their limited target domain. These controllers may be invoked very rapidly, but only produce modest changes to ensure quick decision times. Higher level controllers have increasingly larger workload bands, longer times between invocation (e.g., hourly, daily, weekly), larger sets of more potent actions to choose from, more hosts and applications to consider, and correspondingly take longer to make their decisions.

Figure 36 illustrates the architecture of a single controller. The architecture consists of a set of "predictor modules" and an "optimizer module." The predictor modules, which include the Performance Manager, the Power Consolidation Manager, the Cost Manager, and the Workload predictor (ARMA filter), use analytical models described in Chapter 2.4 to predict utility values of new configurations and actions being considered by the optimizer. Given a configuration c and workload W, the Performance and Power Consolidation Managers predict the corresponding application utility and power utility values. When provided with a list of actions in addition to c and W, the Cost Manager predicts the action costs, while the ARMA filter uses previous workload history to predict the stability intervals. The optimizer module includes the Optimal Adaptation Search engine and is responsible for choosing the optimal set of actions that will maximize the utility. The search is guided

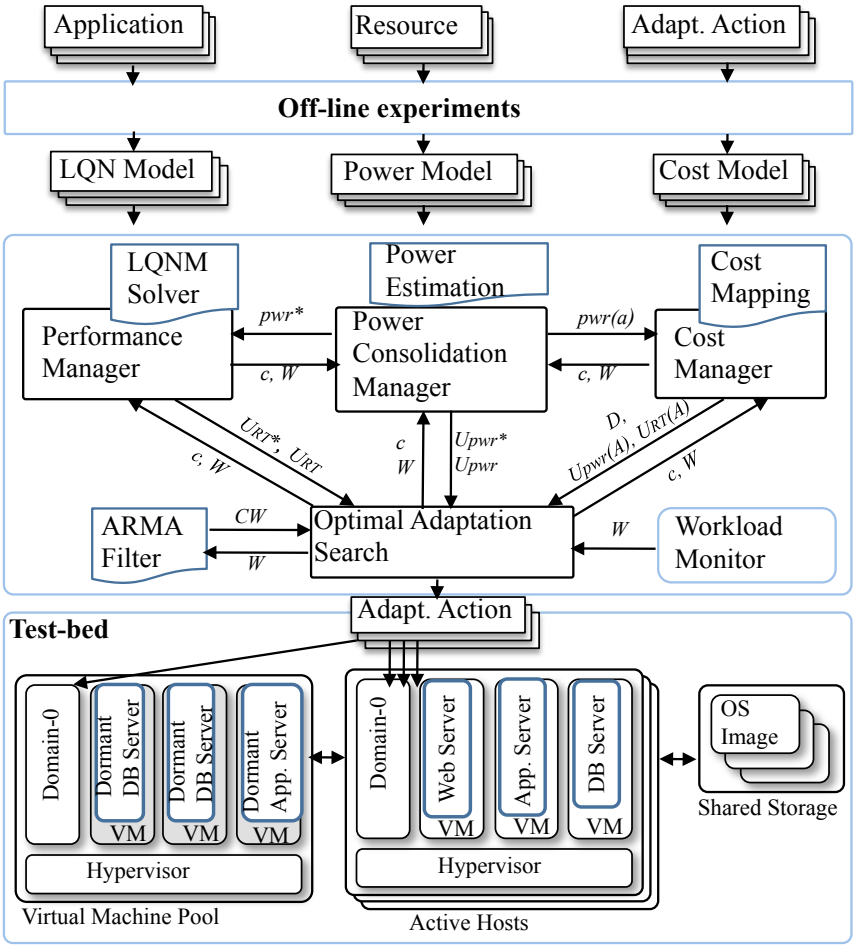


Figure 36: Architecture of a single controller

by upper bounds of utility estimates (denoted by the superscript "*") which are provided by the predictor modules.

5.2.2 Performance and Power Optimizer

The multi-dimensional optimization approach relies on a simpler optimizer, Perf-Pwr optimizer, to provide the best configuration while ignoring any adaptation costs. In this section, the Perf-Pwr optimizer is described.

Perf-Pwr optimizer generates the optimal tradeoff between performance and power consumption for a given workload when any transient adaptation costs are ignored. It is extended from the performance optimizer presented in Chapter 3 that finds the configuration

that maximizes utility given a fixed pool of resources (and ignoring power usage). Specifically, Perf-Pwr finds the optimal capacities of VMs that can be packed on as few server machines as possible while balancing performance and power usage. Similar to our prior performance optimizer, Perf-Pwr optimizer employs a heuristic bin-packing algorithm to place given VMs to hosts and a classic gradient-based search algorithm, but extends the algorithm to deal with variable number of active hosting machines and their power consumption.

Perf-Pwr optimizer determines the optimal configuration (i.e., one that maximizes application utility) first for the whole system (all hosts active) and then reduces the number of hosts to see if a smaller number of active hosts would optimize overall utility (application utility + power usage). Specifically, for any given set of hosts, Perf-Pwr optimizer initially allocates maximum CPU capacities for all VMs. Then, it attempts to place (pack) these VMs on the given set hosts (bins). The bin-packing algorithm used by the optimizer chooses the host that has the largest space among used hosts. If no such host is found, it chooses a new empty host only if it is available. If the bin packing fails, the optimizer starts a search process where, in each iteration, it generates a set of candidate configurations by (a) reducing the capacity of individual VMs and (b) reducing the replication level of an application component (and thus, removing one VM). Then, it chooses the candidate that has the highest *gradient* value among all the candidates, where the gradient is defined as $\nabla\rho = (\rho^{new} - \rho)/(U_{RT}^{new} - U_{RT})$ and ρ^{new} and U_{RT}^{new} represent each new candidate's CPU utilization and performance utility, respectively. It attempts to pack the chosen candidate c_i on the given set of hosts and if the packing fails, the algorithm performs the next iteration using configuration c_i as the new starting point. If the packing succeeds, the optimizer considers the resulting configuration as a *potential optimal configuration* and repeats the search with a reduced number of hosts. For each potential optimal configuration, the optimizer estimates watts consumed by each host by summing all hosted VMs' utilization and also shared utilization (i.e., consumed by Domain-0). The optimizer stops reducing number

of hosts when the number of hosts reaches a threshold that can host minimum capacities of the VMs. The potential optimal configuration that has the largest utility is chosen as the "ideal configuration" c^* and its utility denotes the "ideal utility" $U^* = U^*_{RT} - U^*_{pwr}$. The ideal utility is an upper bound for the multi-dimensional optimization since it ignores adaptation costs.

5.2.3 Multi-Dimensional Optimizer

The optimization algorithm incorporates power and performance overheads caused by adaptation actions and the cost of the decision making process into the tradeoff formulation. Given the utility function and models, the adaptation system determines the optimal sequence of adaptation actions that transforms the current configuration c to the new optimal configuration. To solve the multi-dimensional optimization, the A* search algorithm present in Chapter 4.2.2 is adopted. To recall the search algorithm, this section briefly outlines it and then, describes the improved algorithm to deal with the cost of the decision making process.

Naive A* algorithm. The A* algorithm requires a "cost-to-go" heuristic to be associated with each vertex of the graph. This heuristic estimates the shortest distance from the vertex to a goal state and, for the result to be optimal, the heuristic must be "permissible" in that it overestimates the cost-to-go. We use the ideal utility U^* as the heuristic since it represents the highest utility that can be generated for the given workload. Since it does not consider any costs, it overestimates the utility and therefore, it is a permissible heuristic.

The algorithm starts from v_0 with current configuration. In each iteration of search, it generates the set of child vertices as one adaptation step from a parent vertex (e.g.,v_0) and stores these vertices in the total set of explored vertices V. It also stores the parent vertex only if it is a candidate configuration. It then chooses the vertex v from V with the lowest utility. Each vertex's utility is computed by summing the cost of actions from v_0 plus the cost-to-go if the vertex is an intermediate, or the total utility U if the vertex

is a candidate. If the chosen vertex v is a candidate, the algorithm returns the vertex and computes actions. The algorithm considers v as the final optimal configuration since the vertex's utility is larger than any other utilities of intermediate configurations that can be generated by further explorations. This is because those utilities (i.e., the cost-to-go plus accumulated cost) will decrease as further actions are taken. Meanwhile, utilities of any other candidates generated by further explorations are less than those utilities generated with cost-to-go by the definition of permissible. Thus, optimality is guaranteed.

Since the naive A* algorithm still evaluates a large number of configurations due to the numerous possible adaptation actions at each depth of the graph, the search time may increase exponentially as the number of hosts and applications increases. For example, if the workload changes significantly and then stays relatively long in this state (resulting in a large control window), the algorithm may try to change the current configuration significantly by searching a large number of possible action sequences. The huge search space, and the resulting long search time, is a general problem for many optimization techniques proposed in the literature for cloud computing environments. Spending too much time to compute an optimal configuration can adversely affect the system response time (and utility) since the current configuration that may not be optimal for the changed workload is used during the decision making. Furthermore, the optimization procedure itself may consume significant amount of power while making its decision - so called "consuming power to save power." Therefore, we consider the cost of decision as another tradeoff in our optimization formulation.

Self-Aware A* algorithm. We have developed a method to accelerate the search by decreasing the search space at each vertex dynamically (i.e., decreasing the number of adaptation actions considered for each configuration in the graph). We set the algorithm to choose a small portion of all possible expanded configurations based on similarity to the ideal configuration c^*. Specifically, the algorithm computes a weighted Euclidean distances

between each expanded configuration and c^* by summing up the differences in the corresponding VM sizes (CPU capacities) in the two configuration. We also set a weight to each VM based on its relative size in the ideal configuration. For example, we set 2 times more weight to VM_i than VM_j if their CPU capacities are 60% and 30%, respectively, in the ideal configuration. In addition, we compute another distance value based on VM placement on hosts by counting how many VMs have identical locations (same host) in the two configurations and then normalize the value with the total number of VMs.

Our Self-Aware A* algorithm uses the weighted Euclidian distances and a heuristic to dynamically restrict the search space and to allow the optimization system to control the cost of search versus the potential benefits during the search process. Specifically, it measures the elapsed time of the search, T, the utility accrued of the current configuration, U_T, and the power usage of the search procedure itself, U_{pwr^T}. Then, the algorithm compares the cost to an "expected utility", U_H, to decide when the search space needs to be restricted (i.e., search needs to be completed soon). We consider a history of recent utilities and choose the lowest one as U_H (i.e., a pessimistic estimate). Furthermore, we set a delay threshold for the search $\mathcal{T}$ that depends on the length of control window and can be empirically obtained. This threshold prevents a too long search in the case U_H is too high for the current system state. When the cost of search reaches U_H, or T exceeds $\mathcal{T}$, the optimization system accelerates its search by decreasing search width of each vertex. The resulting search algorithm is shown in Algorithm 3. Note that the risk of stopping too early and never finding the correct adaptations is reduced by the fact that the optimization system operates multi-level controllers and lower level controllers will refine the configuration chosen by the higher level controllers.

The algorithm takes the current configuration c, workload W, the length of control window CW, the expected utility U_H, its performance and power utilities over the unit monitoring interval U_{RT^H} and U_{pwr^H}, and the search delay threshold $\mathcal{T}$ as inputs and returns the optimal sequence of adaptation actions A. Using `Perf-Pwr`, the algorithm computes

Input: $c, W, CW, U_H, U_{RT^H}, U_{pwr^H}, \mathcal{T}$ **Output:** A

$(c^*, U_{RT^*}, U_{pwr^*}) \leftarrow$ `Perf-Pwr`(W);
if $c^* = c$ **then return** "$null$";
$v_0.(A, c, U_{RT}(A), U_{pwr}(A), U_{RT}, U_{pwr}, D)$
$\leftarrow (\phi, c, 0, 0, U_{RT^*}, U_{pwr^*}, 0)$;
$(V, T, U_T, U_{pwr^T}, st) \leftarrow (\{v_0\}, 0, 0, 0,$ `Time`$())$;
$(U'_{RT}, U'_{pwr}) \leftarrow$ `UtilityEst` (c, W);
while *forever* **do**
  $v_{max} \leftarrow \text{argmax}_{v \in V} v.U$; $t \leftarrow 0$;
  if $v_{max}.a_{last} =$ "$null$" **then return** $v_{max}.A$;
  foreach $a \in \mathcal{A} \cup$ "$null$" **do**
    $v_n \leftarrow v_{max}$; $v_n.A \leftarrow v_{max}.A \cup a$;
    $v_n.c \leftarrow$ `NewConfig` $(v_n.c, a)$;
    $V_n \leftarrow V_n \cup v_n$;
  if $(U_T + U_{pwr^T}) \geq U_H$ *or* $(T \geq \mathcal{T})$ **then** $V_n \leftarrow$ `PruneByDistance`$_{v_n \in V_n}(v_n.c, c^*)$;
  foreach $v_n \in V_n$ **do**
    if $v_n.c =$ "$candidate$" **then**
      $(U_{RT}, U_{pwr}) \leftarrow$ `UtilityEst` $(v_n.c, W)$;
      $v_n.U \leftarrow (CW - v_n.D) \cdot (U_{RT} - U_{pwr}) + (v_n.U_{RT}(A) - v_n.U_{pwr}(A))$;
    else
      $(d, U_{RT}(a), U_{pwr}(a)) \leftarrow$`Cost` $(v_n.c, W, a)$;
      $v_n.U_{RT}(A) \leftarrow v_n.U_{RT}(A) + d \cdot U_{RT}(a)$;
      $v_n.U_{pwr}(A) \leftarrow v_n.U_{pwr}(A) + d \cdot U_{pwr}(a)$;
      $v_n.D \leftarrow v_n.D + d$;
      $v_n.U \leftarrow (CW - v_n.D) \cdot (U^*_{RT} - U^*_{pwr}) + (v_n.U_{RT}(A) - v_n.U_{pwr}(A))$;
    if $\exists v' \in V$ *s.t.* $v'.c = v_n.c$ **then**
      if $v_n.U > v'.U$ **then** $v' \leftarrow v_n$;
    else
      $V \leftarrow V \cup v_n$;
  $t \leftarrow$ `Time`$() - st$; $st \leftarrow$ `Time`$()$; $T \leftarrow T + t$;
  $U_{pwr^T} \leftarrow U_{pwr^T} + t \cdot U_{pwr^t}$;
  $U_T \leftarrow U_T + t \cdot (U'_{RT} - U'_{pwr})$;
  $U_H \leftarrow U_H - t \cdot (U_{RT^H} - U_{pwr^H})$;

Algorithm 3: Optimal adaptation search

the ideal utilities. The `UtilityEst` estimates performance and power utilities, U'_{RT} and U'_{pwr}, with current configuration and workload. The elapsed time T, the utility accrued by the current configuration U_T, the power consumption incurred by the search procedure itself U_{pwr^T}, and expected utility U_H are updated after each depth of search.

In each iteration in the `while` loop, the open vertex with the highest utility is selected as v_{max}. If this vertex's configuration is a "candidate" (i.e., its last action is "null"), then the algorithm considers the configuration as the optimal one and returns actions leading to the configuration as described in the Naive A* algorithm. Otherwise, it explores further by triggering all possible actions including "null" (i.e., "do nothing"). `NewConfig` generates a new vertex (configuration) resulting from performing action a in the current vertex. If the cost of the search (i.e., $U_T + U_{pwr^T}$) exceeds the expected utility, or the elapsed time exceeds the given delay threshold, the algorithm prunes the number of new vertices using the Euclidean distances described above by calling `PruneByDistance`. When a new configuration is a "candidate", the algorithm invokes `UtilityEst` to estimate performance and power utilities. Otherwise, it invokes `Cost` to compute the adaptation costs such as accrued performance and power utilities (i.e, $U_{RT}(A)$ and $U_{pwr}(A)$, respectively), and then computes the total utility with the cost-to-go values. If the newly generated vertex v_n is the same as one previously found, say v', and v_n's utility is larger than that of v', then the algorithm replaces the old vertex with the new one.

5.3 Evaluation Results

5.3.1 Experimental Setup

The 3-tier servlet version of RUBiS application used in our prior research work (see Chapter 3.3.1 and 4.3.1) has been deployed to evaluate the multi-dimensional optimization approach. The application workload has been set to remain in the range 0 to 100 req/sec. We have set the maximum replication level for Tomcat and MySQL servers to 2, which is enough to handle the maximum request rates (100 req/sec), while Apache has not been

replicated since a single Apache server per application is enough even under the maximum request rates. To replicate the database server, we have used a simple master-slave mechanism provided by MySQL. All tables have been copied and synchronized between replicas (i.e., shared-nothing). We have deployed up to 4 RUBiS instances, referred to as from RUBiS-1to RUBiS-4, and thereby, deployed up to 20 VMs in a small data center. To simulate the scalability of our approach, we have set up to 12 RUBiS instances and 60 VMs.

The Xen-based virtualization setup has also been reused. In this setup, we have used up to 2 racks, each of which has 6 commodity Pentium-4 1.8GHz machines with 1GB of memory running on a single 100Mbps Ethernet segment. As illustrated by the test-bed box in Figure 36, one machine in each rack is dedicated to host dormant VMs used in server replication, and one as a storage server for VM disk images. We hook all machines to a power meter to measure power usage. Each VM is allocated 200MB of memory with a limit of up to 4 VMs per hosting machine. The remaining 200MB is allocated to Domain-0. The total CPU capacity of all VMs on a hosting machine is capped to 80% to ensure enough resources for Domain-0 even under loaded conditions. We set the minimum CPU capacity for each VM to 20% to avoid request errors even under low request rates. We use Xen's credit-based scheduler to dynamically set CPU capacity of each VM.

In the following experiments, we have generated up to 4 different workloads based on the Web traces from the 1998 World Cup site and the traffic traces of an HP customer's Internet Web server system. A typical day's traffic has been chosen from each trace. Then, we have scaled and shifted them to the range of request rates that our experimental setup can handle. Specifically, we have scaled both the World Cup request rates of 150 to 1200 req/sec and the HP traffic of 2 to 4.5 req/sec to our desired range of 0 to 100 req/sec. Since our workload is controlled by adjusting the number of simulated clients, we have created a mapping from the desired request rates to the number of simulated concurrent sessions. Figure 37 shows these scaled workloads from 15:00 to 21:30, where workload-1

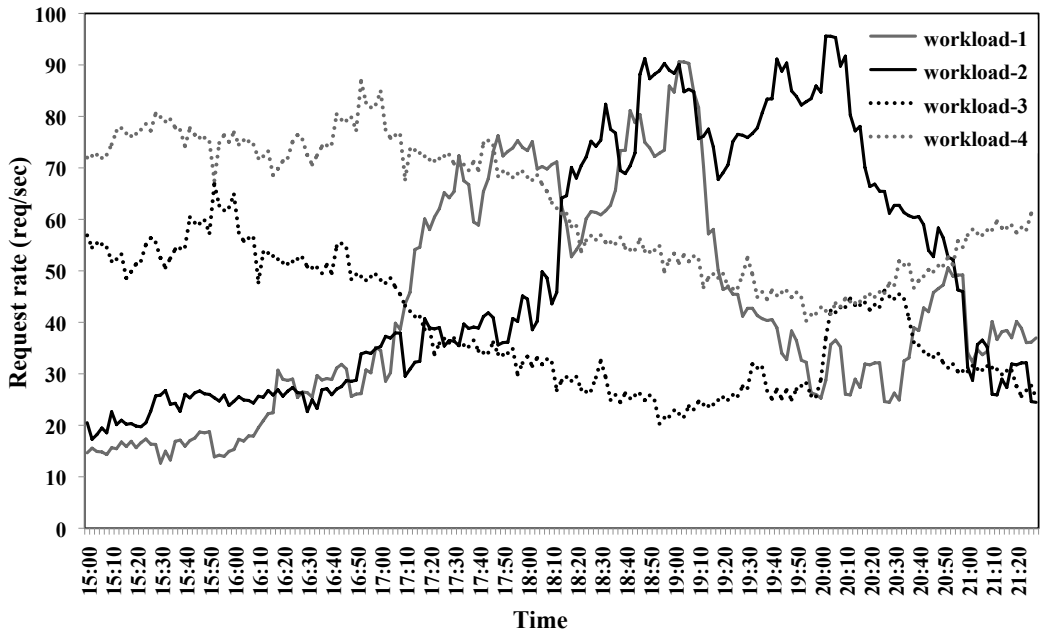
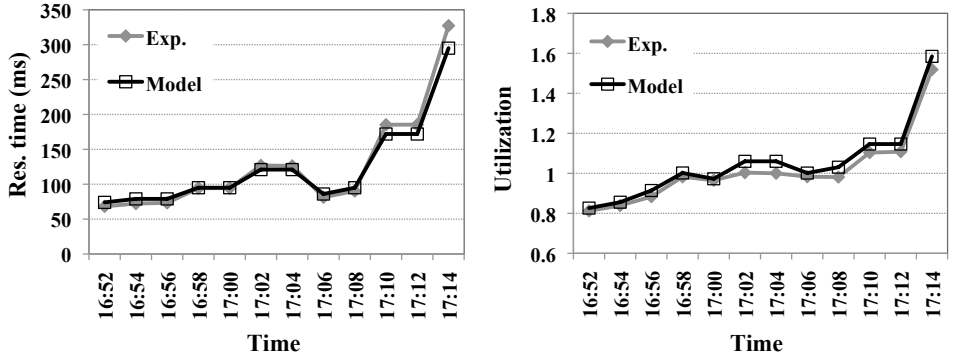


Figure 37: Application workloads

and workload-2 use the scaled World Cup trace, and workload-3 and workload-4 use the HP workload trace.

5.3.2 Experimental Model Validation

This section presents experiment results including the accuracy of models introduced in Chapter 2.4 and various adaptation overheads.



(a) Response times (b) Utilizations

Figure 38: Application model accuracy

Response time and utilization. We have validated the application performance models

using the workload-1 in Figure 37. Figure 38 shows that our performance models (LQNM) provide sufficient accuracy for (a) response time and (b) utilization. The estimation error is approximately 5%. In this model validation, the interval from 16:52 to 17:14 of the workload has been used since it represents the first flash crowd in the scenario as shown in Figure 37. While the Performance Manager generates a series of configurations using models for given request rates, we have recorded estimated response times and CPU utilizations. Then we have compared them with experiment results. In these experiments, we have restarted the controller to measure values at each time point separately for each configuration and request rate to remove any noise caused by adaptations.

Power consumption. To apply the power model introduced in Chapter 2.4.2 into the multi-dimensional optimization, the non-linear model has to be calibrated to fit into actual power consumption observed using a power meter. In the fitting process, we have to measure three model parameters and then, set the tuning parameter used to obtain the non-linearity of the model. Those parameters are pwr_{idle} representing the power consumption of the machine at standby state, pwr_{busy} representing the maximum power consumption of the physical machine observed in our workload scenario, and ρ representing CPU utilization of the machine estimated by the LQN models at the workload.

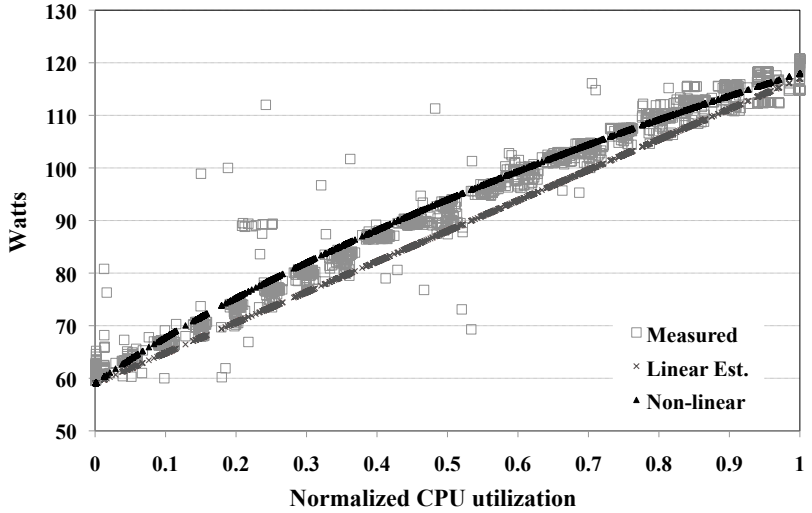


Figure 39: Power model fitting using Ubench

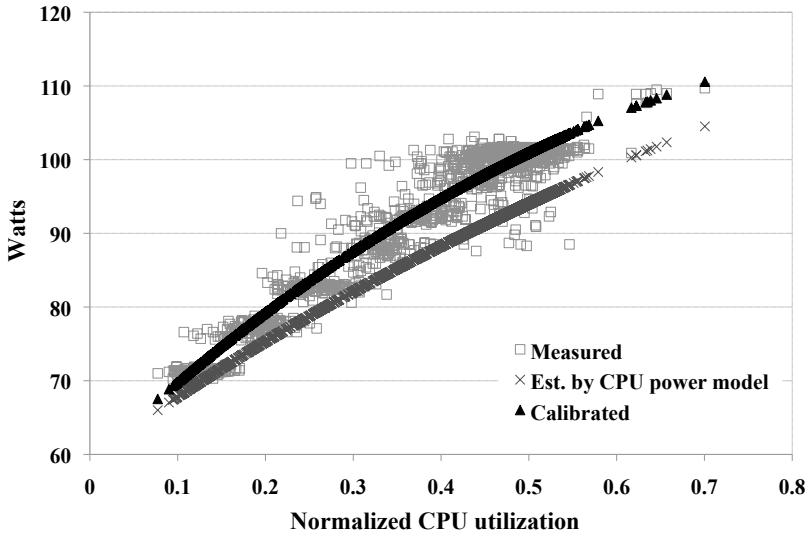


Figure 40: Power model fitting using RUBiS with read-only transaction mix

In our power model, we consider only CPU utilization as the major factor of server power consumption. To obtain a power model based on CPU utilization, we have employed a micro-benchmark, called "Ubench," that intensively consumes CPU resource but little of other resources such as memory, disk, and network bandwidth. Figure 39 shows two models with measured power consumption. To measure the power consumption, we have hooked a machine into the power meter, and then deployed the benchmark on the machine. As shown in the figure, the power consumption has a strong correlation with CPU utilization. We have first applied a linear model by setting the tuning parameter to 1 and then, increased it until the model curve is fitted to measured power consumption. The error rate of the non-linear model is around 2%, while the one of the linear model 8%. Then, we have applied the non-linear model to the RUBiS application with read-only transaction mix workload. Figure 40 shows that the power model obtained with Ubench is not fitted into measured power consumption. This is because the RUBiS application and its workload are more complex than the micro-benchmark, and other resources such as dynamic memory usage may be involved in the server power consumption. Therefore, we have to further calibrate the model by increasing the tuning parameter. The error rate of the calibrated model is around 4%. To measured power consumption, we have increased the

workload (i.e., the number of concurrent users) until CPU utilization reaches around 70 % while measuring power consumption every second. Note that we couldn't measure power beyond 70% utilization, since we have connection errors from the Web server at that point.

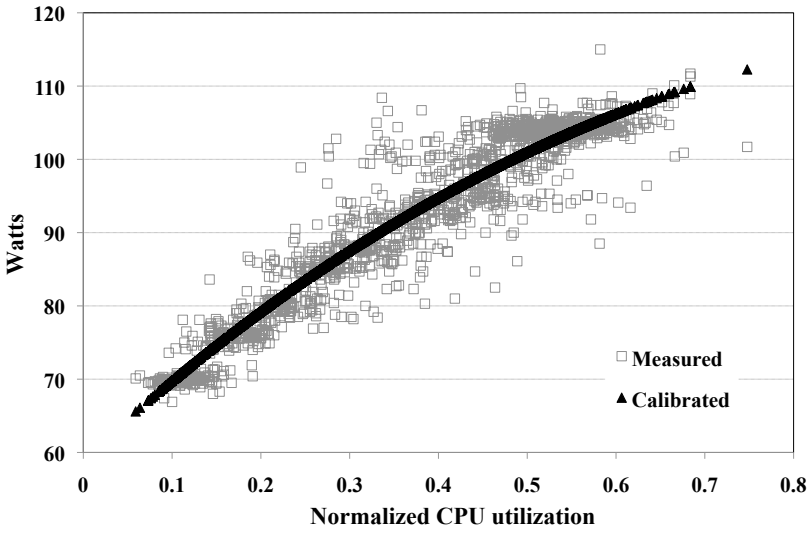


Figure 41: Power model fitting using RUBiS with arbitrary transaction mix

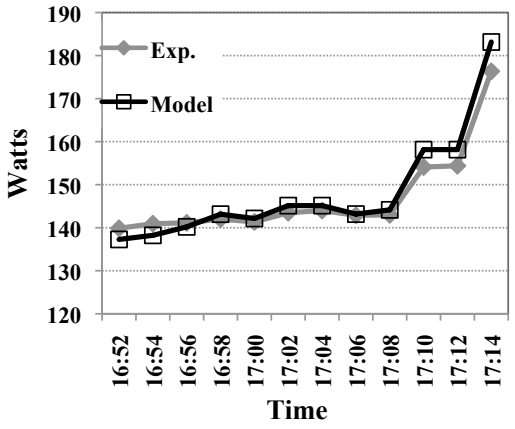


Figure 42: Power model accuracy

To evaluate the accuracy of the calibrated power model, we have applied the model to an arbitrary transaction mix of RUBiS. Figure 41 shows that the model is as accurate as the read-only workload of RUBiS is. We have validated the power model using the same methodology as the response time and utilization accuracy validation. Figure 42

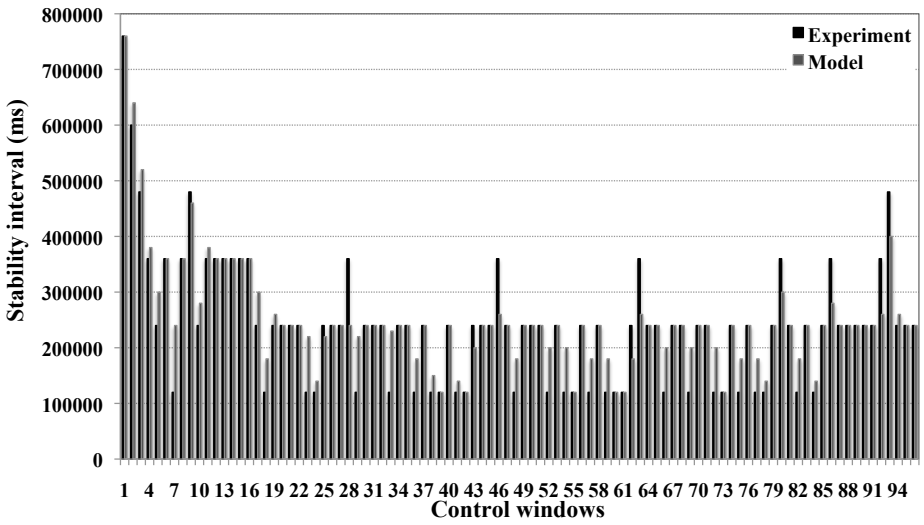


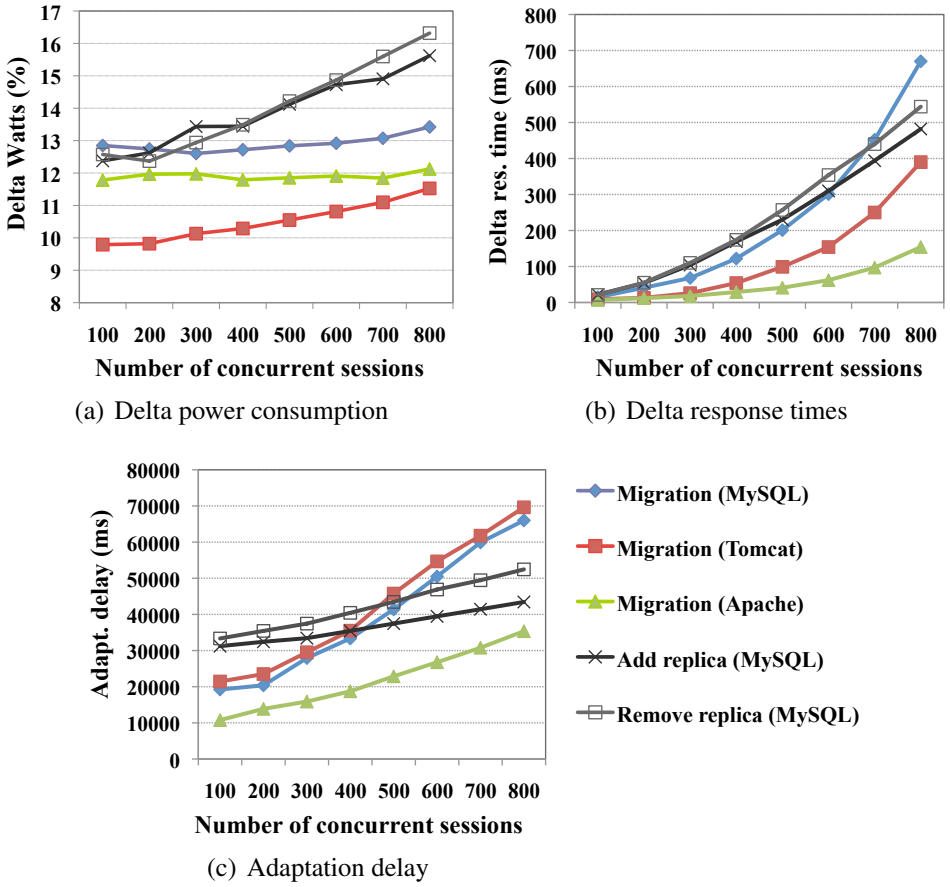
Figure 43: Accuracy of stability interval estimation

shows the model accuracy for power consumption in this scenario. The estimation error is approximately 5%.

Stability interval. We have evaluated the ARMA filter introduced in Chapter 2.4.4 using workload-1 and workload-3 in Figure 37. In this evaluation, we have first measured the number of control windows and the interval of each window under the scenario.

As shown in Figure 43, we have 97 windows ranged from 120 seconds to 760 seconds. Then, we have run the ARMA filter to estimate windows and the interval of each window. Once the workload is suddenly changed, the deviation of estimated interval is relatively high, but stabilized immediately at followed windows. The average error is reasonably small (approximately 14%), so that the ARMA filter can be facilitated in the optimization framework to predict the stability of workload.

Adaptation Costs. Figure 44 illustrates some of the adaptation costs measured for the RUBiS application on the small data center. The figures illustrate that adaptation costs are heavily influenced by the workload, and vary in tier components and adaptation actions. We have also measured power overhead and duration off-line for shutting down/restarting



(a) Delta power consumption

(b) Delta response times

(c) Adaptation delay

Figure 44: Adaptation costs

hosts. Starting a host has taken around 90 sec and consumed around 80 watts while shutdown has taken 30 sec and consumed 20 watts.

5.3.3 Adaptation Comparison

To evaluate our approach, we compare our optimization's results with those of three different approaches, each of which solves the tradeoff between two objectives among performance, power consumption, and adaptation costs as follows:

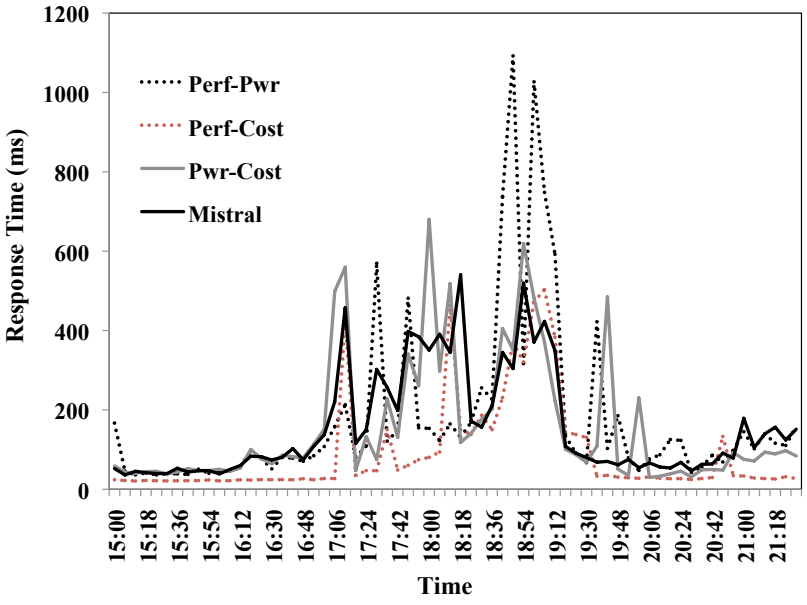
- Perf-Pwr: addresses the tradeoff between performance and power consumption, but ignores transient adaptation costs. Our Perf-Pwr optimizer described in Chapter 5.2.2 is adopted. In this approach, once a workload change is observed in a monitoring interval, the optimizer chooses adaptation actions and executes them.

- Perf-Cost: multiplexes a given fixed pool of resources to hosted applications to maximize performance utility. The cost-sensitive approach presented in Chapter 4 is directly used in this comparison work. This approach incorporates adaptation costs (adaptation duration and performance overhead) into the optimization formulation in each control window. However, it considers neither further power savings by consolidating VMs to a smaller number of hosts, or power overhead during adaptations.

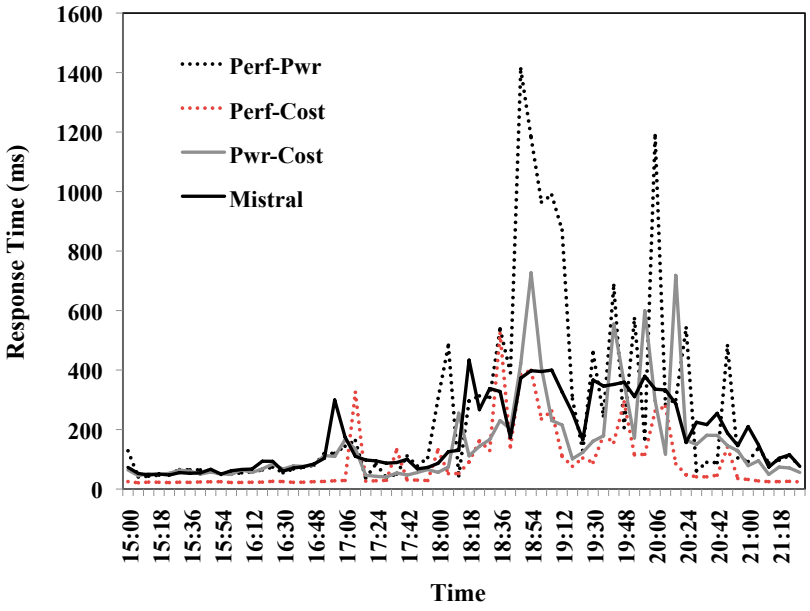
- Pwr-Cost: minimizes power consumption and adaptation costs under the restriction that VMs' CPU capacities and placement for each request rate are given and static. These CPU capacities are large enough that the target response time can be met. To compute such VMs' capacities, the Perf-Pwr optimizer will be modified so that it will not reduce the VM sizes below the capacity needed to meet the target response times. Given these VMs' capacities and placement across the resource hierarchy at each execution, the Pwr-Cost optimizer first adjusts the VMs' capacities of the VMs in the current configuration to match the new sizes. If the resulting VM CPU capacities violate the capacity constraints on some host (the sum of VM capacities on a host must be less than 100%), the VMs are migrated over the resource hierarchy starting from the smallest one until the constraints are satisfied on all host. Finally, when no constraints are violated, the algorithm uses VM migration to consolidate VMs on fewer hosts if possible. During consolidation decision, it considers the tradeoff between power saving through consolidation and migration cost. Compared to our approach, this approach does not allow response time goals to be missed in order to reduce power usage or transient costs.

- Mistral: performs the multi-dimensional optimization for performance, power consumption, and adaptation costs. This represents our approach.

We compare these four approaches using RUBiS-1 and RUBiS-2 deployed in a rack with first and second level controllers. The target response time has been set to 400ms.



(a) RUBiS-1 Response Time



(b) RUBiS-2 Response Time

Figure 45: Response time comparison of adaptation approaches

This time has been derived experimentally as the mean response time across all transactions of the RUBiS application running with a "default configuration" where all tiers' CPU capacities have been set to 40% and workload has been constant at 50 req/sec. The exact amount of the reward and penalty depends on the current application request rate as presented in Chapter 2.1. For the utility function, the monitoring interval is set to 2 minutes so that we can react quickly to workload changes. The cost per watt consumed over a monitoring interval was set to $ 0.01 in our experiments. We set the workload band to 0 req/sec for the 1^{st}-level controller and 8 req/sec for the 2^{nd}-level controller to ensure that even relatively small workload changes could cause the controller to be triggered.

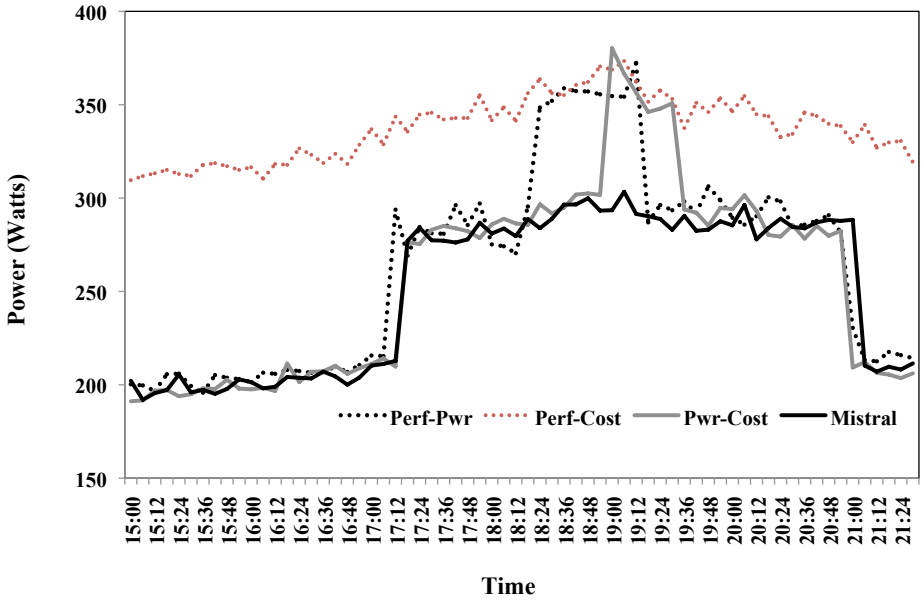


Figure 46: Power consumption comparison of adaptation approaches

Figures 45 through 47 show the results of the comparison. As demonstrated by Figure 45 (a) and (b), the response times with Mistral (i.e., our multi-dimensional approach) are somewhat higher than with the Perf-Cost approach, and it slightly violates the performance objective when request rates peak since it uses a maximum of 3 hosts out of the 4 to save power. However, due to more frequent and intensive adaptations in other approaches (shown as spikes in figures), performance violations with Perf-Pwr and Pwr-Cost are more

frequent than with Mistral. Especially, the response times with Perf-Pwr fluctuate and then remain high since it performs many more adaptations than Mistral. Pwr-Cost is forced to execute migrations during the peak request rates to meet the capacity constraints since it does not address the tradeoff between performance and costs as Mistral has done.

Meanwhile, the overall power consumption with Mistral is lower than with the other approaches as illustrated in Figure 46. This is because Mistral uses fewer hosts and performs fewer adaptations even under peak request rates. The curve of Perf-Pwr shows that using 4 hosts at peak request rates provides the optimal tradeoff between performance and power. However, Mistral chooses configurations with only 2 or 3 hosts since the cost of using 4 hosts would be too high. Pwr-Cost, however, is forced to use 4 hosts when both applications' request rates peak in order to host all the VMs with the required VM CPU capacities.

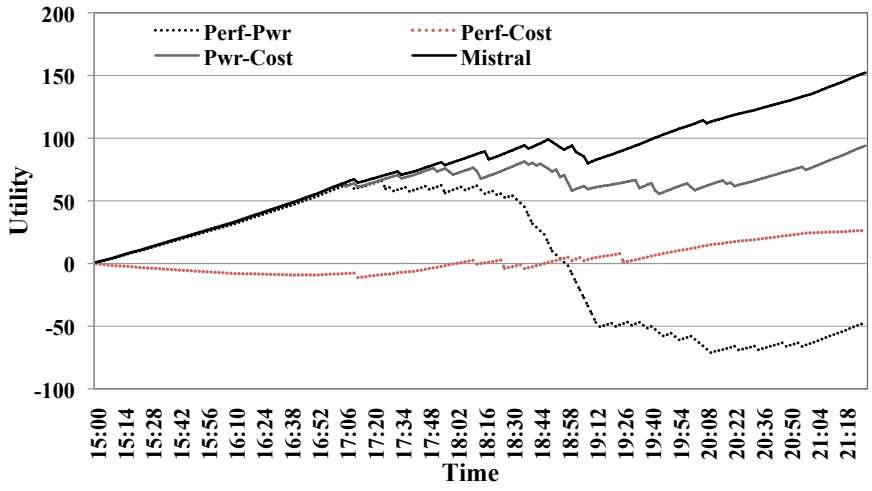


Figure 47: Cumulative utility

Finally, Figure 47 shows that the total utility of Mistral is indeed higher than the other approaches. For the duration of the experiment, the cumulative utility of Mistral (152.3) is higher than those of Perf-Pwr (-47.1), Perf-Cost (26.3), and Pwr-Cost (93.9). Although Perf-Cost provides a better response time behavior than Mistral, its utility is much lower

than Mistral's since it consumes much more power. Thus, Mistral meets the goal of maximizing overall utility, consisting of performance and power utilities and transient costs, better than the other approaches. The results of comparisons are summarized in Table 8.

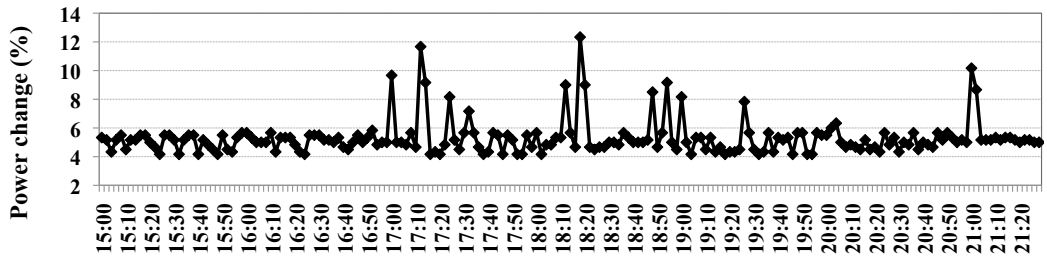
Table 8: Summary of comparison

	Perf-Pwr	**Perf-Cost**	**Pwr-Cost**	**Mistral**
The number of violations	21	6	10	8
Cumulative violations (ms)	10,357.4	385.9	1,960.8	401.2
Cumulative power (Watts)	17,402.1	27,276.4	17,160.4	16,584.1
Cumulative utility	-47.1	26.3	93.9	152.3

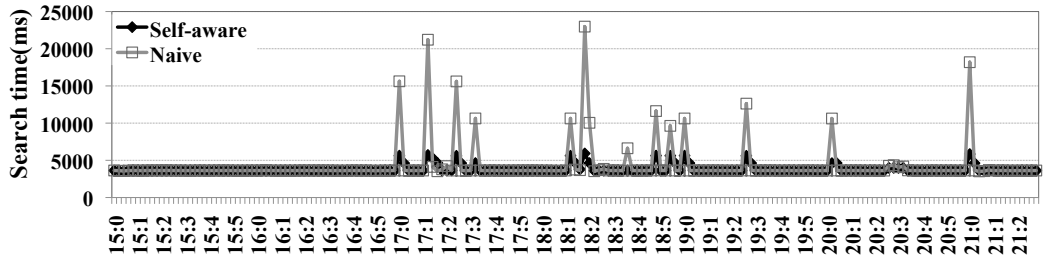
5.3.4 Cost of Search

In this section, we illustrate the cost of the decision making itself in terms of its power consumption, duration, and impact on the total utility. Specifically, we demonstrate that our Self-Aware search algorithm that is aware of its own execution costs can indeed result in significant improvement of overall utility. To measure the power consumption of search algorithms, we connected only the host running controller to the power meter and then ran the controller in a simulation mode where it only determines the action sequences to execute, but does not execute the adaptation actions chosen. Figure 48 (a) shows that the naive search algorithm consumes power up to 12 % over the host's idle power usage (i.e., 60 watts).

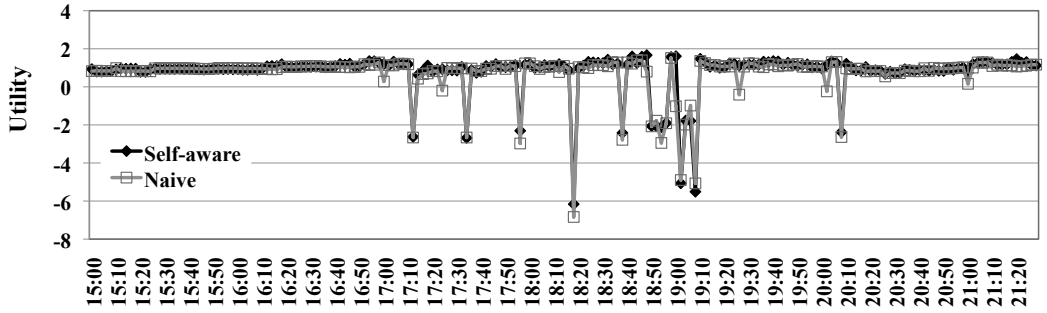
The next two experiments measured how the awareness of its own execution costs impacts the search algorithms. Figure 48(b) shows that the execution time of the naive search approach is up to 4 time longer (around 24 sec) than that of the Self-Aware search algorithm (around 5.5 sec) in the most intensive search cases. The longer search not only uses more power, but also keeps the system in the current configuration, which is not necessary close to optimal for the current workload, a longer time when the search for new configuration is in progress.



(a) Power consumption



(b) Duration



(c) Utility

Figure 48: Cost of search

Finally, we show that such cost awareness does indeed improve the total utility. Figure 48 (c), based on a 2-application scenario (RUBiS-1 and RUBiS-2), shows that the utility of the naive approach is lower than that of the Self-Aware approach, although the Naive approach typically executes more adaptation actions. The difference in the cumulative utilities over the execution time period is significant, with cumulated utilities of 135.3 (Naive) and 152.3 (Self-Aware).

5.3.5 Scalability of Search

Finally, we demonstrate how Mistral scales to the larger number of applications and hosts, and discuss its use in managing large-scale data centers. In the small data center setup, we

Table 9: Search durations and utilities

	2-app	**3-app**	**4-app**
#VMs / #hosts	10 / 4	15 / 6	20 / 8
Self-Aware (avg. duration, milli-sec)	3,807.8	5,669.9	7,514.8
- 1^{st} level	112.4	298.1	487.4
- 2^{nd} level	3,737.6	4,977.2	5,956.8
- 3^{rd} level	5,287.4	8,029.7	10,797.4
Naive (avg. duration, milli-sec)	4,341.4	11,343.4	35,155.8
- 1^{st} level	132.5	355.4	792.5
- 2^{nd} level	4,077.5	5,798.7	11,615.9
- 3^{rd} level	13,387.2	59,345.6	250,297.4
Mistral (total utility)	152.3	336.6	504.8
Ideal (total utility)	351.7	538.3	701.9

deploy up to 20 VMs of 4 RUBiS applications to all given 8 hosts in 2 racks, each of which has 4 hosts. Then, we deploy each RUBiS to the rack for the 2-app scenario. For the 3-app scenario, we add RUBiS-3 to the first rack and for the 4-app scenario, we add RUBiS-4 to the second rack. We configure Mistral to use up to three-level controller. Each 1^{st} level controller can manage a host using CPU tuning, and each 2^{nd} level controller can manage a subset of hosts using CPU tuning, VM migrations, and VM replications within its managed rack. The 3^{rd} level controls the whole system, and uses all actions used in those 1^{st} and 2^{nd} level controllers across hosts and racks. We deploy a controller for each rack to manage the rack and its all hosts, instead of deploying each controller per host in this experiment, while we separately deploy the 3^{rd} level controller to manage the whole resource infrastructure.

Table 9 summarizes those results of 3 different scenarios. We report the average search times for the Naive A* and the Self-aware controllers as well as the averages for each level's controllers. As the number of hosts and applications increase, the search space of adaptation actions to be considered increases. The search duration of the Naive A* search algorithm illustrates the exponential increase. To tackle this problem, our Self-aware search algorithm restricts the search space when necessary using simple technique based on weighted Euclidean distances. The results show that the duration for the Self-aware

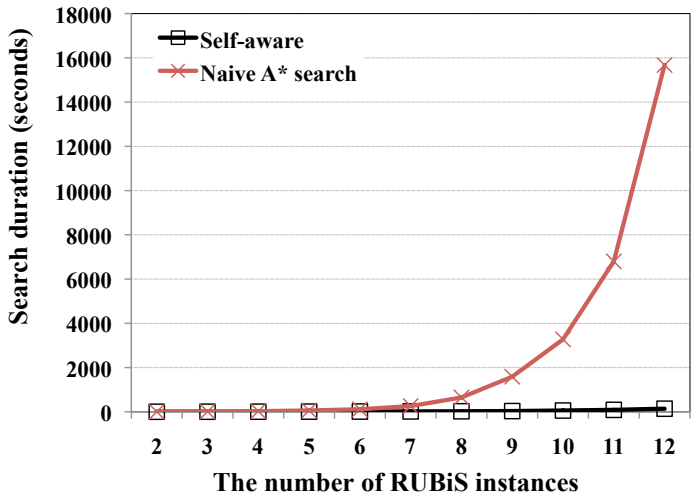


Figure 49: Scalability of search algorithm

algorithm increases approximately linearly with the number of machines, while generating reasonable utilities. To estimate the optimality of our approach, we compare these utilities to the ideal utilities generated by the simulated Perf-Pwr optimizer that ignores adaptation costs. The results show that the gap between the achieved and ideal utilities in each scenario remains approximately constant. It indicates that our approach keeps generating near optimal utilities.

To show further scalability of our search algorithm, we have simulated larger data center by putting up to 12 RUBiS instances and deploying these instances into up to 60 VMs and 24 hosts. In this experiment, we assume that each RUBiS instance can use up to 5 VMs and 2 hosts. Hence, when we add a RUBiS instance, we add 5 VMs and 2 hosts into the search setup. We have measured the average search duration of all controllers (i.e., from 1^{st} level to 4^{th} level) at each point in Figure 49. The figure demonstrates that the Self-aware algorithm can dramatically reduce the search duration. When 12 RUBiS instances are deployed to 48 hosts, the search duration of the Self-aware algorithm is 122 seconds, while it takes almost 4 hours in the Naive search algorithm. These results have implications on using Mistral to manage an entire data center or a cloud platform.

Another feature of our approach to support the scalability is the hierarchical adaptation mechanism. Centralized optimization techniques are typically not scalable enough to manage a large system. Mistral can address this challenge due to its ability to implement multi-level hierarchical control. Specifically, each local controller managing its own few machines (e.g., a stack of machines in a rack) can execute frequently (every few minutes), while higher-level controllers can operate hourly or daily on larger groups of machines (e.g., the whole data center hosting a number of racks).

5.4 Work Related to Multi-Dimensional Optimization

In this section, we present recent work most pertinent to the discussion of this dissertation in the field of the power management and performance-power optimization. We categorize related work based on adaptation methodologies and objectives.

Many efforts have tackled intelligent power control using underlying hardware support such as processor throttling and low-power DRAM states. In particular, Dynamic voltage and frequency scaling of processors (DVFS) has been adopted by many authors including [60, 18, 26, 52], but mainly in single-server settings. Extending these control algorithms to balance end-to-end performance across multiple tiers and clusters remains a challenge. Nevertheless, we believe that techniques such as DVFS are complementary to our approach and can be incorporated into the lowest level controllers.

Some researchers have worked to maximize the use of a given power budget across multiple machines and tiers. In [25], the authors present a methodology for efficient power provisioning that increases the number of services that can be deployed within a given power budget. Govindan et al. have tackled a similar problem using statistical multiplexing methods to improve the power utilization in [34]. However, they do not explicitly consider the power-performance tradeoff nor any transient costs.

A number of projects have addressed different aspects of the power-performance tradeoff. Gandhi et al. use queuing models to find the optimal power allocation among servers

so as to minimize mean response time under a given power budget in [30], while Kephart et al. address the tradeoff in [47] using reinforcement learning over a decentralized architecture in which power and performance managers cooperate. Chase et al. discuss turning servers on and off for efficient power management in [16]. However, these approaches have not considered adaptation costs which, as we have shown, can have a significant impact on overall utility. Similar to our approach, Chen et al. consider some adaptation costs (time overheads and wear-and-tear) [18], but they do not consider process migration and consolidation.

The authors of [56] propose a technique that exploits the hypervisor's ability to limit hardware usage of VMs and control power consumption of individual VMs in a fine-grained manner. The mechanism can be integrated with our approach as an adaptation action to achieve further power savings at an aggregated level.

Recently, a number of power management systems have been proposed based on virtualization techniques, including [51, 70, 75, 12, 50], that share some adaptation methods with our approach. Tolia et al. demonstrate the ability of such techniques to optimize the performance-power tradeoff in two case studies using COTS hardware [70]. Cardosa et al. control min, max, and share parameters of VMs to manage the power-performance tradeoff and develop constrained bin-packing algorithms [12]. However, they do not consider benefits and costs of VM migrations that can be used to further consolidate servers by packing VMs into a smaller number of physical machines in such virtualized environments.

Kusic et al. tackles a similar problem of achieving power efficiency while maintaining the desired performance by consolidating servers, and also explicitly deals with transient costs [51]. While they consider the potential excessive costs caused by high workload variations in their problem formulation, they only consider a single type of adaptation (turning on/off machines). Moreover, adding multiple actions to their approach is not trivial and can lead to significant challenges with scalability.

The pMapper system [75] tackles power-cost tradeoffs under a fixed performance constraint by using modified bin-packing algorithms to minimize migration costs while packing VMs in a small number of machines. Our Pwr-Cost approach is inspired by pMapper. Similarly, Sanjay et al. perform VM placement to save power without degrading performance [50]. They also consider adaptation costs to improve system stability in their distributed architecture. However, their focus is on developing an extensible architecture to coordinate various management objectives, rather than solving tradeoffs between those objectives.

5.5 Summary

Managing large computer systems (e.g., data centers, clouds) with complex multi-tier distributed applications is becoming increasing important and challenging due to often conflicting goals of meeting performance objectives, saving power, and managing the cost of management decisions and actions. In this work, we have presented the multi-dimensional optimization approach, a control architecture that optimizes total utility that includes application utility due to meeting/missing performance objectives, power costs, and transient adaptation costs. We demonstrate experimentally that our adaptation engine, referred to as Mistral, provides better overall utility than a number of alternative controllers that consider only a subset of these factors. To our knowledge, our self-aware search algorithm is the first one to consider the cost of the search itself in its decision making. We demonstrate experimentally that such self-awareness does indeed improve overall total utility. Mistral can also be configured as a multi-level hierarchical controller enabling its potential application in large scale systems.

CHAPTER VI

CONCLUSION AND DISCUSSION

6.1 Conclusion

This dissertation addresses the problems associated with dynamic resource management in cloud computing infrastructures. Although cloud computing based on resource virtualization has provided significant features for improving power efficiency while meeting performance requirements in enterprise data centers, it has also posed new research challenges. One critical challenge regards how to dynamically provision available resources to maximize overall utility under time-varying workloads, while minimizing management costs. A satisfactory approach to meet the challenge must deal with very different responsiveness of different applications, handle dynamic changes in resource demands as their workloads change over time, and consider management costs such as power consumption and adaptation overheads.

In this dissertation, I have presented research efforts in attacking the challenges. I have developed an adaptation engine to address the problem of the multi-dimensional optimization between performance, power consumption, and transient costs incurred by various adaptations and decision-making process itself in a cloud infrastructure. This research effort is based upon the observation that adaptations can cause significant overheads not only on end-to-end performance but also on power consumption. Moreover, such transient overheads can vary in intensity and time scale against workload and performance characteristics of hosted applications. The innovative multi-dimensional optimization approach uses an analytical modeling technique, a scalable optimization search algorithm, and an adaptation hierarchy to deal with large-scale cloud infrastructures and various adaptation

actions in support of various time-scale decision-makings. The approach has been demonstrated in a virtualized data center environment. This research makes the following distinct contributions.

- Transparent and tractable optimization for application performance. The adaptation system constructs adaptation rule set off-line for a given range of workload to be used as a guideline for runtime adaptation decisions. Additionally, domain experts or management systems can be allowed to further inspect or extend the generated rule set with additional constraints such as the consideration of adaptation overheads. It provides the upper-bound performance utility for a given workload that is used in our multi-dimensional optimization process.

- Ability to address the tradeoff between adaptation benefit and cost. Although the resource management is easier than ever before along with virtualization technology, the indiscriminate use of adaptations such as VM migration can have adverse impact on satisfying response-time-based SLAs and power savings. Using modeling techniques, utility functions, and novel optimization algorithms, our approach can address the tradeoff between accrued adaptation costs and their benefits to maximize the overall utility.

- Scalable and efficient optimization for the multi-dimensional tradeoffs. The proposed adaptation system is designed to balance multiple management objectives including performance, power consumption, and transient adaptation costs incurred by adaptation actions and the decision making procedure itself. By developing the self-aware optimization algorithm and the multi-level hierarchical adaptation architecture, we can deal with a large number of applications, hosts, and various adaptation actions at multiple time-scales while solving the optimization problem.

6.2 Future Work

Although we have showed the feasibility of solving the large-scale multi-dimensional optimization problem in this dissertation, we introduce two distinct challenges that need to be further explored in future work.

First, we have mainly focused on CPU resource and end-to-end response time to evaluate our approach. The approach can be extended to allow more complex utility functions incorporating various metrics and statistics, and to allow management of multiple types of heterogenous resources (e.g., memory, disk I/O, and network bandwidth in addition to CPU capacity). Since the adaptation engine uses models to evaluate the utility function, the main consideration in tackling the first challenge is the types of metrics and statistics that can be predicted by the models. Without modification, the models employed in this dissertation can predict response time, throughput, CPU utilization, disk utilization, and I/O throughput. They can also be extended to predict network bandwidth. However, the complexity of modeling various metrics can pose the inherited performance degradation of the approach, and more importantly, need a risk management due to potential inaccurate predictions. Therefore, we need to develop more robust and reliable models to deal with multiple heterogenous metrics. Additionally, in order to manage multiple heterogenous resource types, the bin-packing algorithm used by our adaptation engine must be extended to generate component placements with additional constraints. In particular, to allow for resources with different capacities, one can use one of several approximation algorithms such as a heuristic vector bin packing algorithm for the variable sized bin packing problem. Meanwhile, if a statistic other than the mean value of a metric is required (e.g., fraction of requests for which response time is greater than some threshold), then the models have to be solved by simulation to get accurate answers. For example, the LQNS tool suite used in evaluations of our queueing network models provides a simulator that can produce higher order statistics. When simulation is needed, an off-line solution such as one described in Chapter 3 for performance optimization can become the practical approach to

solve the multi-dimensional optimization problem, since simulation is usually not practical in a purely on-line approach.

As another on-going work, we are integrating other management objectives including service availability and reliability into the optimization formulations. We have introduced a preliminary result for service availability optimization and the tradeoff between availability and performance in [43]. As presented in this dissertation, when multiple management objectives derived from different administrative domains are controlled by a unified optimization system, all potential tradeoffs must be considered. For example, to achieve high service availability, a high redundancy level of each hosted application is typically required. However, it can lead to consuming additional power by imprudently deploying many application replicas. Additionally, to maintain a certain replication level of each application, the management system should address the tradeoff between the cost of replications and service availability by dynamically determining not only how to replicate (e.g, where each replica is placed) but also when replicating applications.

REFERENCES

[1] "Server and data center energy efficiency," in *U.S. Environmental Protection Agency Report*, ENERGY STAR Program, 2007.

[2] "Green clouds: Power consumption as a first order criterion?," in *Panel meeting in the* 17^{th} *International Conference on Autonomic Computing*, 2009.

[3] AMAZON, "Elastic compute cloud," (http://www.amazon.com/ec2/).

[4] APPLEBY, K., FAKHOURI, S., FONG, L., GOLDSZMIDT, M., KRISHNAKUMAR, S., PAZEL, D., PERSHING, J., and ROCHWERGER., B., "Oceano sla based management of a computing utility," in *Proceedings of Symposium on Integrated Network Management*, pp. 855–868, IFIP/IEEE, 2001.

[5] ARLITT, M. and JIN, T., "Workload characterization of the 1998 world cup web site," in *HP Labs Technical Report*, 1999.

[6] AT&T, "Synaptic hosting," (http://www.business.att.com/enterprise/family/application-hosting-enterprise/synaptic-hosting-enterprise/).

[7] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., and WARELD, A., "Xen and the art of virtualization," in *Proceedings of* 19^{th} *Symposium on Operating Systems Principles*, pp. 164–177, ACM, Oct. 2003.

[8] BENNANI, M. and MANESCE, D., "Resource allocation for autonomic data centers using analytic performance models," in *Proceedings of the* 2^{nd} *International Conference on Autonomic Computing*, pp. 217–228, IEEE, June 2005.

[9] BOX, G., JENKINS, G., and REINSEL, G., *Time Series Analysis: Forecasting and Control*. Prentice Hall, 3 ed., 1994.

[10] BUYYA, R., YEO, C. S., and VENUGOPA, S., "Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities," in *Proceedings of the* 10^{th} *International Conference on High Performance Computing and Communications*, IEEE, Aug. 2008.

[11] CAO, J., ANDERSON, M., NYBERG, C., and KIHL, M., "Web server performance modeling using an m/g/1/k*ps queue," in *Proceedings of the* 10^{th} *International Conference on Telecommunications*, pp. 1501–1506, IEEE, Feb. 2003.

[12] CARDOSA, M., KORUPOLU, M., and SINGH, A., "Shares and utilities based power consolidation in virtualized server environments," in *Proceedings of the* 11^{th} *International Symposium on Integrated Network Management*, IFIP/IEEE, 2009.

[13] CEAPARU, I., LAZAR, J., BESSIERE, K., ROBINSON, J., and SHNEIDERMAN, B., "Determining causes and severity of end-user frustration," *International Journal of Human-Computer Interaction*, vol. 17, no. 3, pp. 333–356, 2004.

[14] CECCHET, E., CHANDA, A., ELNIKETY, S., MARGUERITE, J., and ZWAENEPOEL, W., "Performance comparison of middleware architectures for generating dynamic web content," in *Proceedings of the* 4^{th} *International Middleware Conference*, ACM/IFIP/USENIX, 2003.

[15] CHANDRA, A., GONG, W., and SHENOY, P., "Dynamic resource allocation for shared data centers using online measurements," in *Proceedings of SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp. 300–301, ACM, June 2003.

[16] CHASE, J., ANDERSON, D., and THAKAR, P., "Managing energy and server resources in hosting centers," in *Proceedings of the* 18^{th} *Symposium on Operating Systems Principles*, pp. 103–116, ACM SIGOPS, 2001.

[17] CHEKURI, C. and KHANNA, S., *On Multidimensional Packing Problems, 33(4):837 851.* SIAM J. Comput., 1984.

[18] CHEN, Y., DAS, A., QIN, W., SIVASUBRAMANIAM, A., WANG, Q., and GAUTAM, N., "Managing server energy and operational costs in hosting centers," in *Proceedings of SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp. 303–314, ACM, 2005.

[19] CHEN, Y., IYER, S., LIU, X., MILOJICIC, D., and SAHAI, A., "Sla decomposition: Translating service level objectives to system level thresholds," in *Proceedings of the* 4^{th} *International Conference on Autonomic Computing*, p. 3, IEEE, June 2007.

[20] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., and WARFIELD, A., "Live migration of virtual machines," in *Proceedings of the* 2^{nd} *Symposium on Networked Systems Design and Implementation*, pp. 273–286, USENIX Association, 2005.

[21] COFFMAN, E. G., GALAMBOS, G., MARTELLO, S., VIGO, D., DU, D., and PARADALOS, P., *Handbook of Combinatorial Optimization, Bin Packing Approximation Algorithms: Combinatorial Analysis*. Kulwer, 1998.

[22] CUNHA, I., ALMEIDA, J. V., and SANTOS, M., "Self-adaptive capacity management for multi-tier virtualized environments," in *Proceedings of the* 10^{th} *Symposium on Integrated Network Management*, pp. 129–138, IEEE, May 2007.

[23] DILLEY, J., "Web server workload charaterization," in *HP Technical Report, HPL-96-160*, 1996.

[24] DOYLE, R., CHASE, J., ASAD, O., JIN, W., and VAHDAT, A., "Model-based resource provisioning in a web service utility," in *Proceedings of the* 4^{th} *Symposium on Internet Technologies and Systems*, USENIX, Mar. 2003.

[25] FAN, X., WEBER, W., and BARROSO, L., "Power provisioning for a warehouse-sized computer," in *Proceedings of the* 34^{th} *International Symposium on Computer Architecture*, pp. 13–23, ACM, 2007.

[26] FELTER, W., RAJAMANI, K., RUSU, C., and KELLER, T., "A performance-conserving approach for reducing peak power consumption in server systems," in *Proceedings of the* 19^{th} *Annual International Conference on Supercomputing*, pp. 293–302, 2005.

[27] FRANKEN, L. and HAVERKORT, B., "The performability manager," *IEEE Network*, vol. 8, pp. 24–32, Jan. 1994.

[28] FRANKS, G., MAJUMDAR, S., NEILSON, J., PETRIU, D., ROLIA, J., and WOODSIDE, M., "Performance analysis of distributed server systems," in *Proceedings of the* 6^{th} *International Conference on Software Quality*, pp. 15–26, Oct. 1996.

[29] GALLETTA, D., HENRY, R., MCCOY, S., and POLAK, P., "Web site delays: How tolerant are users?," *Journal of the Assoc. for Information Systems*, vol. 5, no. 1, pp. 1–28, 2004.

[30] GANDHI, A., HARCHOL-BALTER, M., DAS, R., and LEFURGY, C., "Optimal power allocation in server farms," in *Proceedings of SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ACM, 2009.

[31] GARBACKI, P. and NAIK, V. K., "Efficient resource virtualization and sharing strategies for heterogeneous grid environments," in *Proceedings of Symposium on Integrated Network Management*, pp. 40–49, IFIP/IEEE, 2007.

[32] GMACH, D., ROLIA, J., CHERKASOVA, L., BELROSE, G., TURICCHI, T., and KEMPER, A., "An integrated approach to resource pool management: Policies, efficiency and quality metrics," in *Proceedings of International Conference on Dependable Systems and Network*, pp. 326–335, IEEE, 2008.

[33] GOOGLE, "App engine," (http://code.google.com/appengine/).

[34] GOVINDAN, S., CHOI, J., URGAONKAR, B., SIVASUBRAMANIAM, A., and BALDINI, A., "Statistical profiling-based techniques for effective power provisioning in data centers," in *European Conference on Computer Systems*, pp. 317–330, ACM, 2009.

[35] GOVINDAN, S., NATH, A., DAS, A., URGAONKAR, B., and SIVASUBRAMANIAM, A., "Xen and co.: Communication-aware cpu scheduling for consolidated xen-based hosting platforms," in *Proceedings of the* 3^{rd} *International Conference on Virtual Execution Environments*, pp. 126–136, ACM, June 2007.

[36] HP, "Openview," (http://h18013.www1.hp.com/products/servers/management/openview/).

[37] IBM, "Tivoli," (http://www-01.ibm.com/software/tivoli).

[38] JACOBSON, P. A. and LAZOWSKA, E. D., "The method of surrogate delays: Simultaneous resource possession in analytic models of computer systems," *ACM SIGMETRICS Performance Evaluation Review*, vol. 10, pp. 165–174, Sept. 1981.

[39] JUNG, G., HILTUNEN, M., JOSHI, K., SCHLICHTING, R., and PU, C., "An offline approach for generating online policies," in *The* 8^{th} *International Workshop on Performability Modeling of Computer and Communication Systems*, 2007.

[40] JUNG, G., JOSHI, K., HILTUNEN, M., SCHLICHTING, R., and PU, C., "An off-line approach for generating on-line adaptation policies in consolidated server environments," *Under review in ACM Transactions on Autonomous and Adaptive Systems*.

[41] JUNG, G., JOSHI, K., HILTUNEN, M., SCHLICHTING, R., and PU, C., "Generating adaptation policies for multi-tier applications in consolidated server environments," in *Proceedings of the* 5^{th} *International Conference on Autonomic Computing*, pp. 23–32, IEEE, June 2008.

[42] JUNG, G., JOSHI, K., HILTUNEN, M., SCHLICHTING, R., and PU, C., "A cost-sensitive adaptation engine for server consolidation of multi-tier applications," in *Proceedings of the* 10^{th} *International Middleware Conference*, pp. 163–183, ACM/IFIP/USENIX, Nov. 2009.

[43] JUNG, G., JOSHI, K., HILTUNEN, M., SCHLICHTING, R., and PU, C., "Performance and availability aware regeneration for cloud based multitier applications," in *Proceedings of the* 40^{th} *International Conference on Dependable Systems and Network*, IEEE/IFIP, 2010.

[44] JUNG, G., PU, C., and SWINT, G., "Mulini: An automated staging framework for qos of distributed multi-tier applications," in *Proceedings of Workshop on Automating Service Quality*, pp. 10–15, ACM, Nov. 2007.

[45] JUNG, G., SWINT, G., PAREKH, J., PU, C., and SAHAI, A., "Detecting bottleneck in n-tier it applications through analysis," in *Proceedings of the* 17^{th} *Distributed Systems: Operations and Management*, pp. 149–160, Springer, Oct. 2006.

[46] KANG, J. and PARK, S., "Algorithms for the variable sized bin packing problem," *European Journal of Operational Research*, vol. 147, pp. 365–372, June 2003.

[47] KEPHART, J., CHAN, H., DAS, R., LEVINE, D., TESAURO, G., RAWSON, F., and LEFURGY, C., "Coordinating multiple autonomic managers to achieve specified power-performance tradeoffs," in *Proceedings of the* 4^{th} *International Conference on Autonomic Computing*, pp. 24–33, IEEE, 2007.

[48] KHANNA, G., BEATY, K., KAR, G., and KOCHUT, A., "Application performance management in virtualized server environments," in *Proceedings of the* 10^{th} *Network Operations and Management Symposium*, pp. 373–381, IEEE, 2006.

[49] KOH, Y., KNAUERHASE, R., BOWMAN, M., WEN, Z., and PU, C., "An analysis of performance interference effects in virtual environments," in *Proceedings of International Symposium on Performance Analysis of Systems and Software*, pp. 200–209, IEEE, Apr. 2007.

[50] KUMAR, S., TALWAR, V., KUMAR, V., RANGANATHAN, P., and SCHWAN, K., "vmanage: Loosely coupled platform and virtualization management in data centers," in *Proceedings of the* 6^{th} *International Conference on Autonomic Computing*, pp. 127–136, IEEE, 2009.

[51] KUSIC, D., KEPHART, J., HANSON, J., KANDASAMY, N., and JIANG, G., "Power and performance management of virtualized computing environments via lookahead control," in *Proceedings of the* 5^{th} *International Conference on Autonomic Computing*, pp. 3–12, IEEE, June 2008.

[52] LEFURGY, C., WANG, X., and WARE, M., "Power capping: A prelude to power shifting," *Cluster Computing*, vol. 11, pp. 183–195, May 2008.

[53] LIU, L., WANG, H., LIU, X., JIN, X., HE, W., WANG, Q., and CHEN, Y., "Greencloud: a new architecture for green data center," in *Proceedings of the* 6^{th} *International Conference on Autonomic Computing*, pp. 29–38, IEEE, June 2009.

[54] LIU, T., KUMARAN, S., and LUO, Z., "Layered queuing models for enterprise javabean applications," in *Proceedings of the* 5^{th} *International Conference on Enterprise Distributed Object Computing*, pp. 174–178, IEEE, Sept. 2001.

[55] MICROSOFT, "Hyper-v," (http://www.microsoft.com/hyper-v-server/en/us/default.aspx).

[56] NATHUJI, R. and SCHWAN, K., "Virtualpower: Coordinated power management in virtualized enterprise systems," in *Proceedings of the* 21^{st} *Symposium on Operating Systems Principles*, p. 265 278, ACM SIGOPS, 2007.

[57] PADALA, P., HOU, K., SHIN, K., ZHU, X., UYSAL, M., WANG, Z., SINGHAL, S., and MERCHANT, A., "Automated control of multiple virtualized resources," in *Proceedings of the* 4^{th} *European Conference on Computer Systems*, pp. 13–26, ACM SIGOPS, 2009.

[58] PADALA, P., SHIN, K., ZHU, X., UYSAL, M., WANG, Z., SINGHAL, S., MERCHANT, A., and SALEM, K., "Adaptive control of virtualized resources in utility computing environments," in *Proceedings of the* 2^{nd} *European Conference on Computer Systems*, pp. 289–302, ACM SIGOPS, June 2007.

[59] PAREKH, S., GANDHI, N., HELLERSTEIN, J., TILBURY, D., JAYRAM, T., and BIGUS, J., "Using control theory to achieve service level objectives in performance management," *Real-Time Systems*, vol. 23, pp. 127–141, July 2002.

[60] PERING, T., BURD, T., and BRODERSEN, R., "The simulation and evaluation of dynamic voltage scaling algortithms," in *Proceedings of International Symposium on Low Power Electronics and Design*, pp. 76–81, 1998.

[61] PU, C., SAHAI, A., JUNG, G., PAREKH, J., BAE, J., CHA, Y., GARCIA, T., IRANI, D., LEE, J., and LIN, Q., "An observation-based approach to performance characterization of distributed n-tier applications," in *Proceedings of the* 10^{th} *International Symposium on Workload Charaterization*, pp. 161–170, IEEE, 2007.

[62] RAGHAVENDRA, R., RANGANATHAN, P., TALWAR, V., WANG, Z., and ZHU, X., "No power struggles: Coordinated multi-level power management for data center," in *Proceedings of the* 13^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 48–59, 2008.

[63] RUBIS, "Rice University Bidding System," (http://rubis.objectweb.org/.).

[64] RUSSELL, S. J. and NORVIG, P., *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2003.

[65] SALESFORCE, "Enterprise cloud platform," (http://www.salesforce.com/platform/).

[66] SHA, L., LIU, X., LU, Y., and ABDELZAHER, T., "Queueing model based network server performance control," in *Proceedings of the* 23^{rd} *Real-Time Systems Symposium*, pp. 81–90, IEEE, Dec. 2002.

[67] SLOTHOUBER, L., "A model of web server performance," in *Proceedings of the World Wide Web Conference*, June 1996.

[68] SWINT, G., JUNG, G., PU, C., and SAHAI, A., "Automated staging for built-to-order application systems," in *Proceedings of the* 10^{th} *Network Operations and Management Symposium*, pp. 361–372, IEEE, Apr. 2006.

[69] TESAURO, G., JONG, N., DAS, R., and BENNANI, M., "A hybrid reinforcement learning approach to autonomic resource allocation," in *Proceedings of the* 3^{rd} *International Conference on Autonomic Computing*, pp. 65–73, IEEE, June 2006.

[70] TOLIA, N., WANG, Z., MARWAH, M., and BASH, C., "Delivering energy proportionality with non energy-proportional systems - optimizing the ensemble," in *Proceedings of the* 1^{st} *USENIX Workshop on Power Aware Computing and Systems*, 2008.

[71] UDUPI, Y. B., SAHAI, A., and SINGHAL, S., "A classification-based approach to policy refinement," in *Proceedings of the* 10^{th} *International Symposium on Integrated Network Management*, pp. 785–788, IEEE, May 2007.

[72] URGAONKAR, B., PACIFICI, G., SHENOY, P., SPREITZER, M., and TANTAWI, A., "An analytical model for multi-tier internet services and its applications," in *Proceedings of SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp. 291–302, ACM, June 2005.

[73] URGAONKAR, B., SHENOY, P., CHANDRA, A., and GOYAL, P., "Dynamic provisioning of multi-tier internet applications," in *Proceedings of the* 2^{nd} *International Conference on Autonomic Computing*, pp. 217–228, IEEE, June 2005.

[74] URGAONKAR, B., SHENOY, P., CHANDRA, A., GOYAL, P., and WOOD, T., "Agile dynamic provisioning of multi-tier internet applications," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 3, pp. 1–39, Mar. 2008.

[75] VERMA, A., AHUJA, P., and NEOGI, A., "pmapper: Power and migration cost aware application placement in virtualized systems," in *Proceedings of the* 9^{th} *International Middleware Conference*, pp. 243–264, ACM/IFIP/USENIX, 2008.

[76] VMWARE, "Virtualcenter," (http://www.vmware.com/support/vc11/doc/c15alarms.html).

[77] VOORSLUYS, W., BROBERG, J., VENUGOPAL, S., and BUYYA, R., "Cost of virtual machine live migration in clouds: A performance evaluation," in *Proceedings of the* 1^{st} *International Conference Cloud Computing*, Dec. 2009.

[78] WANG, Z., ZHU, X., PADALA, P., and SINGHAL, S., "Capacity and performance overhead in dynamic resource allocation to virtual containers," in *Proceedings of the* 10^{th} *Symposium on Integrated Management*, pp. 149–158, IEEE, May 2007.

[79] WEBSITEOPTIMIZATION.COM, "The psychology of web performance," (Accessed Apr 2009. http://www.websiteoptimization.com/speed/tweak/psychology-web-performance/), May 2008.

[80] WEISS, A., "Computing in the clouds," *netWorker*, vol. 11, pp. 16–25, Dec. 2007.

[81] WEKA (http://www.cs.waikato.ac.nz/ml/weka).

[82] WELSH, M. and CULLER, D., "Adaptive overload control for busy internet servers," in *Proceedings of Symposium on Internet Technologies and Systems*, USENIX, 2003.

[83] WOOD, T., SHENOY, P., and VENKATARAMANI, A., "Black-box and gray-box strategies for virtual machine migration," in *Proceedings of Symposium on Networked Systems Design and Implementation*, pp. 229–242, USENIX, 2007.

[84] WOODSIDE, C. M., NERON, E., HO, E. D. S., and MONDOUX, B., "An "active server" model for the performance of parallel programs written using rendezvouz," *Journal of Systems and Software*, pp. 125–132, 1986.

[85] XI, B., LIU, Z., RAGHAVACHARI, M., XIA, C., and ZHANG, L., "A smart hill-climbing algorithm for application server configuration," in *Proceedings of the* 13^{th} *International Conference on World Wide Web*, pp. 287–296, ACM, May 2004.

[86] XU, J., ZHAO, M., FORTES, J., CARPENTER, R., and YOUSIF, M., "On the use of fuzzy modeling in virtualized data center management," in *Proceedings of the* 4^{th} *International Conference on Autonomic Computing*, p. 25, IEEE, June 2007.

[87] ZHANG, L., XIA, C., SQUILLANTE, M., and III, W. N. M., "Workload service requirements analysis: A queueing network optimization approach," in *Proceedings of the* 10^{th} *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, pp. 23–32, IEEE, Oct. 2002.

[88] ZHANG, Q., CHERKASOVA, L., and SMIRNI, E., "A regression-based analytic model for dynamic resource provisioning of multi-tier applications," in *Proceedings of the* 4^{th} *International Conference on Autonomic Computing*, pp. 27–36, IEEE, June 2007.

VITA

Gueyoung Jung is a Ph.D. candidate at School of Computer Science, Georgia Institute of Technology. His research interests lie in the fields of distributed systems, operating systems, and autonomic computing. More specifically, he is involved in the automated management of large-scale distributed systems encompassing capacity planning, deployment, and dynamic adaptive systems. He holds an MS in Computer Science from Georgia Institute of Technology and a BS in Statistics from Inha University, Incheon, Korea. He has earned IT scholarship from Ministry of Information and Communication Republic of Korea during his graduate study. Before joining the graduate program in Georgia Institute of Technology, he has been a lead system developer at LG-CNS and a startup company, Internet Metrix, in Korea for five years.