# Combined Power and Performance Management of Virtualized Computing Environments Using Limited Lookahead Control

A Thesis

Submitted to the Faculty

of

Drexel University

by

Dara Kusic

in partial fulfillment of the

requirements for the degree

of

Doctor of Philosophy

in

Computer Engineering

October 2008

## Dedications

This thesis is dedicated to my husband, Sean, for his endless support and encouragement, and to my mother, father and dear grandmother for never expecting anything less.

## Acknowledgments

The accomplishment of this thesis would have been possible without the patient instruction of my advisor, Dr. Nagarajan Kandasamy. His guidance toward my progress as a research student has been given with generosity and always constructively. I also thank him for his professional advice which has never led me astray.

To the members of my committee, Dr. Harish Sethu, Dr. Steven Weber, Dr. Bruce Char, and Dr. Geoff Jiang, I would like to extend my sincere thanks for donating their time and efforts to the evaluation of this thesis and its preliminary proposal. Their comments have helped to clarify and improve much of this work.

To all the lab mates, faculty, and staff members who have helped to enliven each day with an exchange of greetings and smiles of encouragement, I thank them from the bottom of my heart. It is the people who make up the Electrical and Computer Engineering Department that have made this a welcoming place to complete my studies.

## Table of Contents

## List of Figures

## Abstract

Combined Power and Performance Management of Virtualized Computing Environments Using Limited Lookahead Control

Dara Kusic

Advisor: Nagarajan Kandasamy, Ph.D.

There is growing incentive to reduce the power consumed by large-scale data centers that host online services such as banking, retail commerce, and gaming. Virtualization is a promising approach to consolidating multiple online services onto a smaller number of computing resources. A virtualized server environment allows computing resources to be shared among multiple performance-isolated platforms called virtual machines. By dynamically provisioning virtual machines, consolidating the workload, and turning servers on and off as needed, data center operators can maintain desired service-level agreements with end users while achieving higher server utilization and energy efficiency.

This thesis develops an online resource provisioning framework for combined power and performance management in a virtualized computing environment serving session-based workloads. We pose this management problem as one of sequential optimization under uncertainty and solve it using limited lookahead control (LLC), a form of model-predictive control. The approach accounts for the switching costs incurred while provisioning physical and virtual machines, and explicitly encodes the risk of provisioning resources in an uncertain and dynamic operating environment.

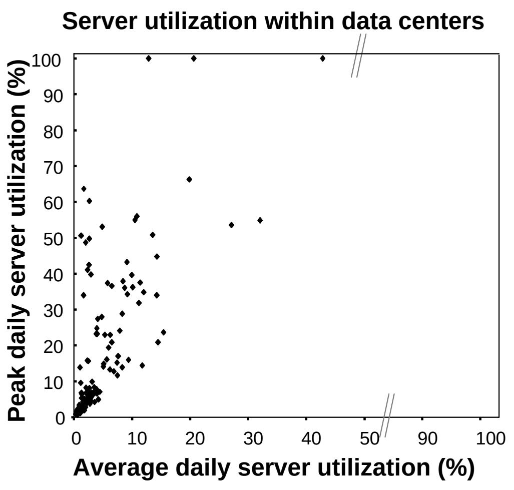
We experimentally validate the control framework on a multi-tier e-commerce architecture hosting multiple online services. When managed using LLC, the cluster saves, on average, 41% in power-consumption costs over a twenty-four hour period when compared to a system operating without dynamic control. The overhead of the controller is low, compared to the control interval, on the order of a few seconds. We also use trace-based simulations to analyze LLC performance on server clusters larger than our testbed, and show how concepts from approximation theory can be used to further reduce the computational burden of controlling large systems.

# 1. Introduction

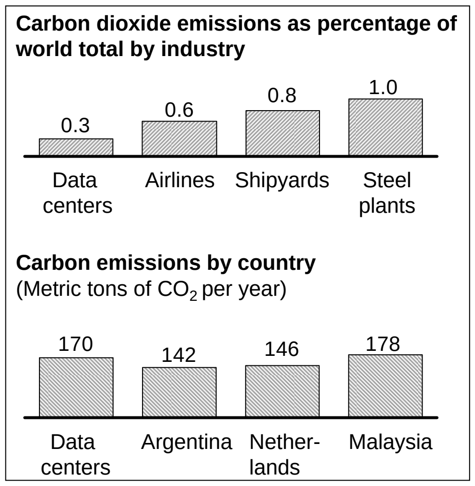
Web-based services such as online banking and shopping must respond continuously to users in real time, and as people have become more reliant on such services, their expectations for dependability and quality of service (QoS) have risen dramatically. Online services are enabled by *enterprise applications*, defined broadly as any software which simultaneously provides services to a large number of users over a computer network. These applications are typically hosted on computer clusters comprising heterogeneous and networked servers, housed in a physical facility called a data center.

A typical data center serves a variety of companies and users, and the computing resources needed to support such a wide range of online services leaves server rooms in a state of "sprawl" with under-utilized resources. Moreover, each new service to be supported often results in the acquisition of new hardware, leading to very low server utilization levels. A recent study by McKinsey & Company, summarized in Fig. 1.1, found that the average server utilization in data centers is only about 6%, and that up to 30% of servers are running idle, that is, the servers are powered on, but perform no useful work [39]. The same study also projects that the world's data centers will surpass the airline industry in carbon dioxide emissions, attributable to their electricity consumption, by 2020. In fact, the carbon dioxide emissions due to data centers currently exceeds the total emissions of some major countries. So, managing the power consumed by these large-scale computing systems is of particular importance, and with energy costs rising about 9% last year [54], there is an urgent need to increase the efficiency of data centers. Although energy conservation tools for computer networks have been under investigation for more than 30 years [42], intelligent energy management solutions have yet to reach maturity.

To operate enterprise computing systems efficiently while maintaining the desired QoS, multiple performance-related parameters must be dynamically tuned to adapt to time-varying



(a)



(b)

Figure 1.1: (a) Average and peak server utilization in data centers, as surveyed by McKinsey & Company. The data shows that up to 30% of servers are idle; they are powered on, but perform no useful work. (b) Carbon dioxide emissions of data centers as compared to other industries and countries.

workload demand and operating conditions. Coping with the numerous performance-related parameters requires systems and applications to become largely *autonomic*, i.e., capable of managing themselves, given only high-level objectives from system engineers, users, and administrators. Key management tasks that can be automated include power management, load balancing, and dynamic resource provisioning (adjusting the allocation of computing resources between multiple services supported by the computing system).

Virtualization is a promising approach to consolidating multiple online services onto fewer computing resources within a data center, thereby reducing power consumption. This technology allows a single server to be shared among multiple performance-isolated platforms called *virtual machines* (VMs), where each virtual machine can, in turn, host multiple enterprise applications. Virtualization also enables *on-demand* or *utility* computing, a dynamic resource provisioning model in which computing resources such as CPU and memory are made available to applications only as needed and not allocated statically based simply on the peak workload [8].

This thesis develops and implements an online optimization framework for combined power and performance management in a virtualized computing environment. The proposed framework allows system administrators to maintain service-level agreements (SLAs) with clients while achieving higher server utilization and energy efficiency by dynamically provisioning VMs, consolidating the workload, and turning servers on and off as needed. This dynamic resource provisioning scheme complements the more traditional off-line capacity planning process [40].

The specific contributions of this thesis are as follows:

- We pose the power/performance management problem as one of sequential optimization under uncertainty and solve it using limited lookahead control (LLC), a form of model-predictive control. The basic idea here is to solve a multi-objective optimization problem that maximizes the performance objective over a given prediction horizon, and then periodically roll this horizon forward. The LLC approach is applicable to computing systems with non-linear behavior where tuning options must be chosen from a finite set at any given time, such as the number of servers or VMs to power up or down. It also models the various switching and opportunity costs associated with turning servers and VMs on or off. For example, profits may be lost while waiting for a VM and its host to be turned on. Other switching costs include the delay incurred in migrating a VM, and the power consumed while a machine is being powered up or down, and not performing any useful work.

- In an operating environment where the incoming workload is noisy and highly variable, excessive switching activity may actually reduce the revenue generated, especially in the presence of the switching costs described above. Thus, each provisioning decision made by the controller is risky and we explicitly encode risk into the problem formulation using preference functions to order possible controller decisions.

We refer to this approach as risk-aware provisioning.

- We validate the control framework using an experimental setup of nine heterogenous servers supporting three online services, enabled by the Trade6, DVDStore, and RUBBoS applications. The cluster processes a time-varying, session-based workload in which both the number of requests as well as the transaction mix can change dynamically during run time. The control objective then is to maximize the profit generated by this system by minimizing both its power consumption and the number of SLA violations. To achieve this objective, the online controller periodically decides: (1) the number of physical and virtual machines to allocate to each service where the VMs and their hosts are turned on or off according to workload demand; (2) the CPU share to allocate to each VM; and (3) how to consolidate VMs, if necessary, by migrating live virtual machines between hosts.

- We use trace-based simulations to analyze controller performance on server clusters larger than our testbed. We show how concepts from approximation theory can be integrated within the LLC framework to further reduce the computational burden of controlling large systems. We use a neural network (NN) to learn the tendencies of the controller, in terms of its decision-making, via offline simulations. At run time, given the current state and environment inputs, the NN provides an approximate solution, which is used as a starting point around which to perform a quick local search to obtain the final control decision.

Our experimental results show that the server cluster, when managed using the proposed LLC approach, saves, on average, 41% in power-consumption costs over a twenty-four hour period when compared to a system operating without dynamic control. The execution-time overhead of the controller is quite low, making it practical for online performance management. We also characterize the effects of different risk-preference functions on control performance, finding that a risk-averse controller reduces the number of SLA violations while maintaining energy savings. A risk-averse controller also reduces server switching activity, a beneficial result when excessive power-cycling of servers is a concern.

The thesis is organized as follows:

Chapter 2 provides some background pertaining to the dynamic resource provisioning problem considered in this thesis. We discuss the use of virtualized execution environments to enable on-demand computing in a data center. We then discuss how control theoretic concepts are being applied for managing the performance of computing systems.

Chapter 3 describes our experimental setup, including the physical system architecture and the enterprise applications used in our experiments. We also discuss issues related to synthetic workload generation.

Chapter 4 formulates the control problem and describes the controller implementation. We develop the functional components of the limited lookahead controller and show how to incorporate the notion of risk into the control problem.

Chapter 5 presents both experimental and simulation results evaluating the controller performance. We quantify the effects of varying controller parameters such as prediction-horizon length and risk-preference values. The chapter concludes with a discussion of the effect of workload prediction error on the results.

Chapter 6 shows how the controller, developed in Chapter 4, can be scaled to manage larger systems using concepts from approximation theory.

Chapter 7 proposes extensions to the work presented herein that use online learning to improve control quality when only partially specified system models are available or when the models slowly change over time. Finally, Chapter 8 concludes the thesis.

# 2. Background

This chapter provides some background to the reader on the concepts of virtualization and on-demand computing. The dynamic resource provisioning approach developed in this thesis uses virtualization technology as a tool to realize the on-demand computing model. We also discuss the basic concepts underlying limited lookahead control.

## 2.1 Virtualization and On-demand Computing

*Virtualization technology* is defined as the set of tools that enable the physical and logical isolation of computing resources (CPU, memory, I/O, disk) such that multiple application environments (OS, program data) may be collated on a single host machine, as shown in Fig. 2.1. The virtualization layer allows for dynamic resource provisioning decisions such as creating or deleting virtual machine instances, and distributing the available processing capacity among multiple virtual machines (VMs). Virtualization can increase hardware utilization and lower capital expenditures, power consumption, and operating costs. Virtualization of our cluster is enabled by VMWare's ESX Server and the VMware VirtualCenter [1]. The VMware ESX Server abstracts the server CPU, memory, storage, and networking resources into VMs that appear to the user as stand-alone machines. VMware VirtualCenter provides a centralized management interface for a cluster of ESX servers. The VirtualCenter is used to issue commands, such as setting the CPU share for a VM, and to monitor the status of VMs in the cluster, including their operating state (on or off), CPU utilization, and network activity. The ESX servers and VirtualCenter also provide an application programming interface (API) to support the remote management of VMs using the simple object access protocol (SOAP).

Virtualization provides an ideal environment for *on-demand computing*, a resource pro-

[1]See `http://www.vmware.com/products/product_index.html` for product information.

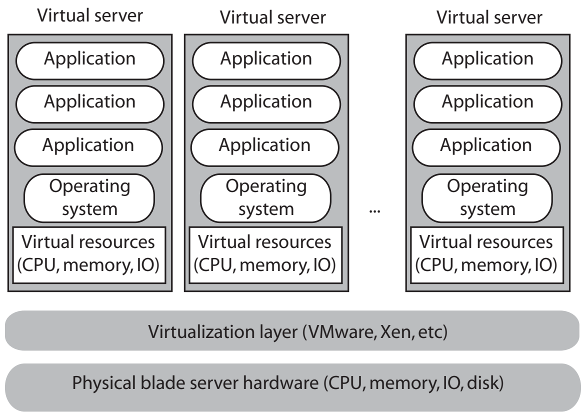

Virtual server
Virtual server
Virtual server
Application
Application
Application
Operating system
Virtual resources (CPU, memory, IO)
...
Virtualization layer (VMware, Xen, etc)
Physical blade server hardware (CPU, memory, IO, disk)

Figure 2.1: A schematic of a physical machine hosting multiple virtual machines using virtualization technology.

visioning model to efficiently support multiple enterprise applications by making computing resources such as CPU share, memory, network I/O, and disk space available to these applications only as needed and not statically allocate them based simply on peak workload demand [8]. The basic provisioning problem in a virtualized environment is to decide an optimal allocation of computing resources among virtual servers under a dynamic workload. Computing resources that can be dynamically provisioned within a virtualized computing environment are as follows.

- **Virtual machines**. A virtual machine may be created from a disk image, or created in advance and simply booted up when additional computing resources are needed.
- **CPU share among multiple VMs**. The processing capacity of a physical machine can be partitioned among one or more VMs in units of MHz. VMs can also be constrained to use only a subset of the CPU cores available on the physical machine.
- **Memory share among multiple VMs**. The memory capacity of a physical machine can be partitioned among one or more VMs in units of MB.
- **Physical servers**. Virtualization provides an interface to consolidate VMs onto a

fewer number of hosts, and to power down host machines when they are not needed.

In addition, virtualization supports migration of VMs, that is, virtual machines may be moved from one host machine to another. Migration is useful for server consolidation, powering down unneeded host machines to save energy, and for re-distributing the workload within a cluster. The migrating VM must use a shared storage device for the disk image, and the CPU, memory and network resources used by that VM shift from one host machine to another. A simple text file contains the configuration options for the VM, such as the number of CPU cores to be used by the VM and its network address, and is passed from one ESX server to another during migration. VM migration is designed to operate transparently to the end user, with no interruption in the services being provided to him/her.

The overall objective of on-demand computing in a virtualized environment is to provision resources to maximize profits while reducing system operating costs, in terms of power consumption. This performance optimization problem may need continuous re-solving with observed environmental events such as time-varying workload arrivals and server failures. Moreover, since workload intensity in enterprise systems can show significant variability within 10s of seconds or a few minutes [59], the system must provision resources over short time scales. Strategies for dynamic resource provisioning often complement off-line capacity planning processes [40, 51, 56].

Use of virtualization technology to assist dynamic rescheduling and consolidation of workloads has been addressed in [25, 55, 56, 61]. Virtualization is used in capacity planning techniques [56] toward the deployment of new computing environments and has been cited as a key strategy in converting conventional data centers to "green" computing environments [60]. Examples of recently introduced deployments of on-demand computing include Amazon's Elastic Compute Cloud (EC2) [48], from which users order computing resources that include network transfer rates, memory, and CPU units for their custom processing needs, and Google Apps [50], which offers users a suite of office applications including email and word processing hosted within large-scale, distributed computing sites.

## 2.2 Autonomic Computing

Managing the numerous control knobs that virtualization exposes is a complex and tedious task, and cannot be performed manually, especially as computing systems continue to expand in scale. In a 2005 poll, more than half of system administrators surveyed said they expected their data centers to grow by as much as 50% within the next five years, and even more said they didn't expect to add any more personnel [5]. As computing systems grow in scale and complexity, existing personnel will become increasingly taxed, and ad hoc and heuristic-based approaches to performance management will become increasingly error prone and ultimately infeasible. Therefore, it is desirable to design on-demand computing systems that can intelligently perform their own resource provisioning, given only high-level objectives from an administrator. A system with such self-management capabilities is referred to as being *autonomic*, and likened to the human nervous system. Autonomic computing systems aim to achieve QoS objectives by adaptively tuning key operating parameters with minimal human intervention [11, 19, 43][2].

As outlined by Kephart and Chess [24], the following properties are desirable in an autonomic computing system:

- **Self-configuration**. The system should configure itself automatically in accordance with high-level objectives. New components introduced into the system should automatically learn about the configuration of the system and behave in a cooperative manner.
- **Self-optimization**. The system should be able to continuously tune itself to improve upon existing performance levels when possible.

[2]Autonomic computing is a term in use since 2001 when IBM management challenged researchers to design scalable and complex computing systems capable of managing themselves [10].

- **Self-healing**. The system should be able to detect, identify, and recover from hardware and software faults that may interrupt service or degrade performance.
- **Self-protection**. The system should defend itself malicious attacks and component failures. It should also anticipate and preempt such compromising events.

This thesis develops the theory and practice of designing autonomic computing systems by applying concepts from control theory to on-demand computing in a virtualized computing environment. In particular, we focus on the topic of *self-optimization,* the property that the system should be able to continuously tune itself to improve upon existing performance levels when possible. The key innovation of the proposed research is in the application of control-theoretic concepts to manage computing systems that exhibit nonlinear behavior and to solve non-convex optimization problems under multiple operating and optimization constraints.

## 2.3 Use of Control Theory for Performance Management

Control theory offers a promising methodology to automate key system management tasks in data centers with some important advantages over heuristic or rule-based approaches. It allows us to solve a general class of problems (including power and performance management) using the same basic control concepts, and to verify the feasibility of a control scheme before deployment on the actual system. Recent research efforts have therefore focused on using concepts from control theory and dynamic programming as the theoretical basis for achieving self-managing behavior in computing applications and systems [2, 17, 22]. Many efforts have focused on using feedback or reactive control as the theoretical basis for autonomic performance management, particularly for problems of task scheduling and CPU provisioning [6, 33, 35], bandwidth allocation and QoS adaptation in web servers [3], load balancing in e-mail and file servers [34, 45], network flow control [38], and CPU power management [36, 53].

Traditional feedback control, however, has some inherent limitations. It assumes a linearized and discrete time model for system dynamics with an unconstrained state space, and a continuous input and output domain. Many practical systems of interest, on the other hand, have the following characteristics.

- **Hybrid behavior**. Many computing systems exhibit behavior comprising both discrete-event and time-based dynamics where the control or tuning options are limited to a finite, discrete set at any given time.
- **Uncertain operating environments**. The workload to be processed is usually time varying and system components may fail during system operation.
- **Multi-variable optimization under constraints**. The cost function specifying application performance requirements can include multiple variables, and must typically be optimized under explicit and dynamic operating constraints.
- **Control actions with dead times**. Actions such as dynamically (de)allocating and provisioning computing resources often incur a substantial dead time (the delay between a control input and the corresponding system response), requiring proactive control where control inputs must be provided in anticipation of future changes in operating conditions.

All of the above characteristics occur in the control problem of interest in our virtualized computing environment. For example, control options must be chosen from a constrained, discrete set, such as the number of VMs to instantiate, and control actions, such as turning on a server, have an associated dead time, usually on the order of minutes. State-space methods adapted from model predictive control [37] and limited lookahead supervisory control [13] offer a natural framework to accommodate these characteristics. They take into account multi-objective non-linear cost functions and dynamic operating constraints while optimizing application performance.

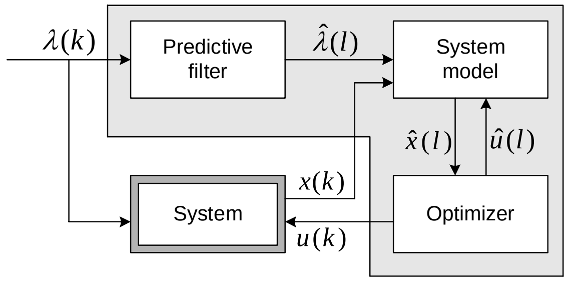


Figure 2.2: The schematic of a limited lookahead controller.

## 2.4 Limited Lookahead Control Concepts

This thesis solves the dynamic resource provisioning problem within a *limited-lookahead control* (LLC) framework. The basic concept is to solve an optimization problem over a future time horizon, and then roll this horizon forward at regular intervals, re-solving the control problem. The LLC scheme allows for multi-objective optimization, explicit constraint handling, and is robust to environmental disturbances.

We can provide an intuitive understanding of lookahead control using driving as an analogy. A driver will usually consider the road several hundred yards ahead to anticipate driving conditions and adjust speed and gear settings accordingly. As the car moves along the road, the driver will see only the next few hundred yards, continuously picking up new information from the limits of this horizon and using it to update control decisions. Lookahead control works similarly: it always considers the predicted system behavior over some time horizon into the future and thus, at each successive sampling instant, predicts one further sample into the future. As new information becomes available, the controller uses it to modify the current trajectory.

The LLC method applies to non-linear systems with dead times where control inputs must be chosen from a finite set within the discrete domain. The LLC approach accommodates cost functions where performance goals can be posed as *set-point regulation* where key operating parameters must be maintained at a specified level (e.g., an average response time in web servers) or *utility optimization* where the system aims to maximize its utility (e.g., the profit-maximization problem considered in this thesis). It is also possible to consider control costs as part of the cost function, indicating that certain trajectories towards the desired goal are preferable over others in terms of their cost to the system.

Fig. 2.2 shows the basic LLC framework. The functional components within the framework can be described as follows.

**Predictive filter**. The LLC framework, as with any predictive control, relies upon environmental forecasting to estimate the system behavior over a trajectory in time. The controller treats time-varying environment inputs to an enterprise computing system—the number of user requests, transaction mix, and the per-request processing time—as disturbances. The controller estimates the disturbances to the system in order to forecast the system behavior. A typical enterprise workload shows pronounced time-of-day variations in which the number of HTTP request arrivals can change considerably in just a few minutes [4, 41]; so accurate predictive filters are essential to ensure good control performance under these operating conditions. In Fig. 2.2, the environment input $\lambda$ is forecast by the

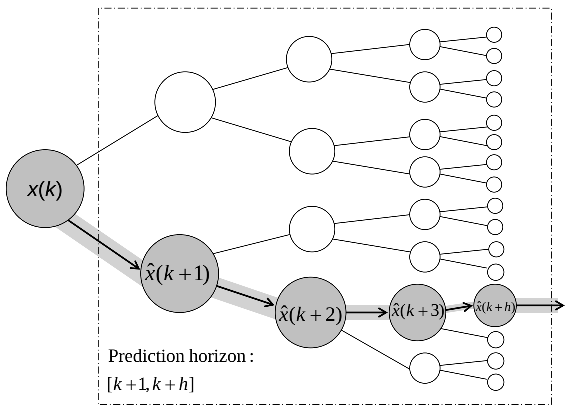


Figure 2.3: The state-space trajectory explored by a limited lookahead controller within a horizon of length $h$. The shaded area shows the best predicted state trajectory $\{\hat{x}(l) | l \in [k+1, k+h]\}$ for a feasible control sequence.

predictive filter over the prediction horizon $h$.

**System model**. Predictions of environmental disturbances, along with current state information $x$ from the system, are passed to the system model. The computing system of interest may be modeled using first-order queueing or difference equations, along with models to estimate power and performance, such as lookup tables for energy consumption and application processing rates, respectively.

**Optimizer**. The optimizer uses the system model to predict a sequence of future system states $\hat{x}$ from time $k+1$ up to a prediction horizon of length $h$, given a sequence control inputs $\{u(l)|l \in [k+1, k+l]\}$ from the feasible set $U$. The objective is then to find a control sequence that minimizes the cumulative cost function $J(\cdot)$ while satisfying both state and input constraints $H(\cdot) \leq 0$ within this horizon, and takes the following general form.

$$\begin{aligned} &\min_{U} \sum_{l=k+1}^{k+h} J\big(x(l), u(l)\big) \\ \text{Subject to:}\quad & H\big(f(x(l), u(l), \hat{\lambda}(l))\big) \leq 0 \\ & u(l) \in U(x(l)) \end{aligned} \tag{2.1}$$

At each time step $k$, the controller performs the optimization in (2.1) to identify a feasible sequence of control actions $\{u^*(l)|l \in [k+1, k+h]\}$ to guide the predicted system state $\{\hat{x}(l)|l \in [k+1, k+h]\}$ within the prediction horizon, as shown in Fig. 2.3. The first input leading to this trajectory is chosen as the next control action, and the rest of the sequence is discarded. The entire process is repeated at time $k+1$ when the new system state $x(k+1)$ is available.

The LLC method is conceptually similar to model predictive control (MPC) [37], widely used in the chemical-process industry to solve optimal control problems. There are, however, some key differences. For example, MPC usually deals with systems operating in a continuous input and output domain whereas LLC targets a discrete domain. Also, since the dynamics of chemical processes vary slowly over time, one can afford to solve computationally expensive MPC problems in reasonable time (compared to the plant dynamics), whereas resource provisioning decisions for enterprise applications must be made quickly due to the rapid variations in workload arrival rate [4, 41].

# 3. Testbed Description

This chapter discusses the setup of our experimental computing cluster, including a description of the three enterprise applications, generation of the enterprise workload, and the multi-tier system architecture hosting the applications.

## 3.1 Introduction

The control framework presented in this thesis is evaluated using both trace-based simulations as well as experiments performed on a computer cluster in our lab at Drexel University. The experimental setup is a cluster of nine heterogenous servers supporting three online services, emulating a small enterprise computing system. The online services process a time-varying, session-based workload in which both the number of requests as well as the corresponding transaction mix can change dynamically during run time. The servers operate in a virtualized environment that enables resources such as CPU, memory, and I/O to be partitioned to each service, on demand, as per changes in workload intensity. The virtualized hosting environment is configured as a two-tier architecture in which the application layer and the data storage layer reside on separate servers. The control objective is to provision resources in the application layer such that the system satisfies the SLA for all three services in the most profitable manner, maximizing revenue and minimizing power consumption, at each time instance.

## 3.2 The Enterprise Applications

We use transaction-based applications for experimental testing in our virtualized hosting environment. These applications provide three online services that we term Gold, Silver, and Bronze.

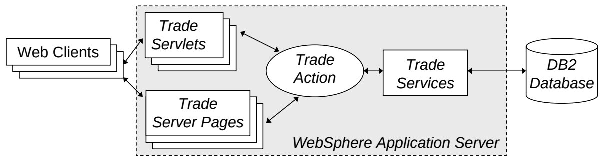


Figure 3.1: The Trade6 stock trading application and related software components. Trade6 is hosted across two physical tiers, the application tier and the database tier, in our experimental testbed.

The *Gold* application is DVD Store [9], an open-source emulation of an e-commerce site hosted on an Apache Tomcat application server with DB2 as the database component. DVD Store has several default database sizes available, and we chose a ‘small’ database size consisting of 20,000 customers and 10,000 products.

The *Silver* application is IBM’s Trade6 benchmark—a multi-threaded online stock-trading service that allows users to browse, buy, and sell stocks [20]. As shown in Fig. 3.1, Trade6 is a transaction-based application integrated within the IBM WebSphere Application Server with DB2 as the database component. This execution environment is then distributed across multiple servers comprising the application and database tiers. We chose the default database size for our experiments, consisting of 500 users and 1,000 traded stocks.

The *Bronze* application is Rice University’s servlet implementation of RUBBoS [52], a bulletin-board benchmark similar to Slashdot with the capability to browse for, and post messages. We host RUBBoS on an Apache Tomcat application server with DB2 as the database component. We chose the default database size for our experiments, consisting of 500,000 users, 12,000 stories, and 2.3 million posted comments.

The above applications operate on the notion of user sessions, since requests issued by a user are dependent on one other. A typical session may proceed as follows. A user logs into the Trade6 online stock trading application with the intent to make a stock purchase. He queries the top-performing stocks of the day and browses the listing. Next, he specifies

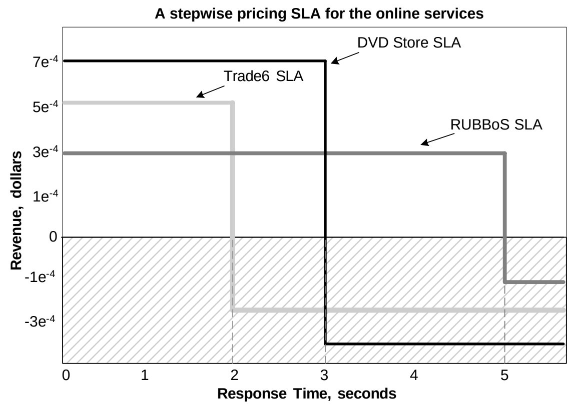


Figure 3.2: A pricing strategy that differentiates the Gold, Silver, and Bronze services. Individual requests below a response time threshold generate revenue for the service, while requests exceeding the threshold result in a penalty paid to the client.

a quantity of stock and sends the purchase requests. He waits for the purchase confirmation before finally logging out of the application. Sessions require that state information be maintained over the duration of the user's interaction.

The three services are differentiated within our revenue model according to a simple pricing scheme. Each request to the Gold, Silver, or Bronze application generates revenue as per the non-linear pricing graph shown in Fig. 3.2 that relates the response time achieved per transaction to a dollar value that the client is willing to pay. Response times below a threshold value result in a reward paid to the service provider, while response times violating the SLA result in the provider paying a penalty to the client. For example, in Fig. 3.2, a response from the Trade6 application within 2 seconds will generate 500 micro-dollars for the system; else, 250 micro-dollars is credited to the client.

## 3.3 The System Architecture

Fig. 3.3 shows the virtualized server environment hosting the three services that are, in turn, distributed over the application and database tiers. A dispatcher balances the incoming workload, with arrival rates $\lambda_1$, $\lambda_2$, and $\lambda_3$, for the Gold, Silver, and Bronze services, re-

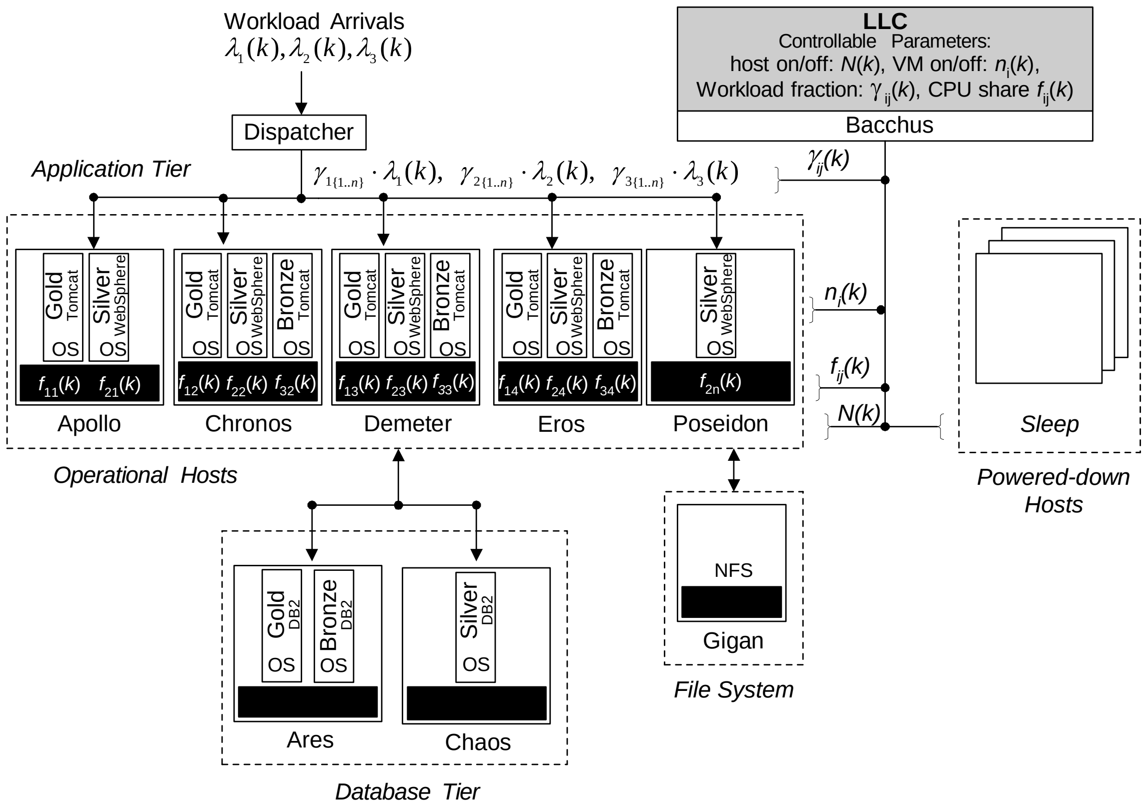


Figure 3.3: The system architecture supporting the Gold, Silver, and Bronze services. The controller sets $N(k)$, the number of active hosts, $n_i(k)$, the number of VMs to serve the $i^{th}$ application, and $f_{ij}(k)$ and $\gamma_{ij}(k)$, the CPU share and the fraction of workload to distribute to the $j^{th}$ VM, respectively. A *Sleep* cluster holds powered-off machines.

spectively, across those VMs running the designated application. Hosts not needed during periods of slow workload arrivals are powered down and placed in the *Sleep* cluster to reduce power consumption. Resource provisioning is performed only within the application tier, and machines in the database tier are never powered down[1].

The nine host machines in the computing cluster, detailed in Fig. 3.4, are networked via a gigabit switch and comprise a heterogeneous rack of seven Dell 2950 and Dell 1950 PowerEdge servers, as well as two Dell desktop machines that have been converted to data servers. Virtualization of this cluster is enabled by VMWare's ESX Server 3.5 running a

[1]There is one database instantiated for each of the Gold, Silver, and Bronze services.

| Host name | CPU Speed | CPU Cores | Memory |
|---|---|---|---|
| Apollo | 2.3 GHz | 8 | 8 GB |
| Ares | 1.6 GHz | 2 | 2 GB |
| Bacchus | 2.3 GHz | 2 | 8 GB |
| Chaos | 1.6 GHz | 2 | 8 GB |
| Chronos | 1.6 GHz | 8 | 4 GB |
| Demeter | 1.6 GHz | 8 | 4 GB |
| Eros | 1.6 GHz | 8 | 4 GB |
| Gigan | 2.3 GHz | 2 | 2 GB |
| Poseidon | 2.3 GHz | 8 | 8 GB |

Figure 3.4: The CPU speeds, number of CPU cores, and memory capacity of the host machines comprising the experimental testbed.

Linux RedHat 3.2 kernel. The operating system on each VM is the SUSE Enterprise Linux Server Edition 10.

The controller, executed on the host Bacchus, aims to meet the SLAs of the Gold, Silver, and Bronze services while minimizing the corresponding use of computing resources at the application tier, in terms of the number of hosts and VMs, and the CPU share per VM. A VM's CPU share is specified as an operating frequency. For example, Chronos, with eight CPU cores, each operating at 1.6 GHz, has $8 \times 1.6 = 12.8$ GHz of processing capacity that can be dynamically distributed among its VMs. The ESX server also limits the maximum number of cores that a VM can use on a host to four, setting an upper bound of 6 GHz for a VM's CPU share, and reserves a total of 800 MHz of CPU share for system management processes. So, on a machine with eight CPU cores, we can, for example, host a 6 GHz Gold VM, a 3 GHz Gold VM, and a 3 GHz Silver VM. Memory is generally the resource limiting the number of VMs a host machine can support. Each VM is reserved 770 MB of RAM, and about 750 MB is reserved for the VMware system management. Thus, a host machine with 4 GB memory can support up to 4 live VMs.

Virtual machine migration using the VMotion feature of VirtualCenter allows a live VM to be moved between hosts at run time, allowing VMs to be consolidated on fewer host machines while preserving the state of active user sessions. VMotion requires that VM disk image files be stored on a network file system, the file server Gigan in Fig. 3.3, while computing resources such as CPU share, memory, and I/O are shifted from one host to another. For example, in Fig. 3.3, the controller may decide to migrate the lone VM on Poseidon to Apollo in order to power down Poseidon and save energy.

Virtual machine (de)instantiation provides another approach to re-configuring the system in Fig. 3.3, should workload intensities change. For example, suppose a predicted increase in the Silver workload cannot be accommodated with the current configuration. The controller will identify the most lightly loaded host, Poseidon in this case, instantiate a VM upon it, and launch the Trade6 application. Conversely, if the Silver workload were predicted to decrease, the controller would shut down the VM on Poseidon and power down the host. Existing sessions on a VM that has been ordered to shut down are re-routed to other VMs running the same application.

In our setup, a VM is configured by the controller to launch a particular service at boot time, but the same VM can be dynamically re-assigned by the controller during run time to host a different service as per workload demand. Fig. 3.3 shows just one sample configuration for the assignment of applications to VMs. The controller may, for example, re-configure one of the live VMs such that Apollo hosts two instances of the Silver application.

## 3.4 Workload Generation

The incoming workload to our testbed is synthetically generated, and has the following characteristics.

- **Session-based requests**. Each user initiates a sequence of requests to the Gold, Silver, or Bronze application that are encapsulated into a user session. Requests for each session are assumed to be dependent upon one each other, that is, the user waits

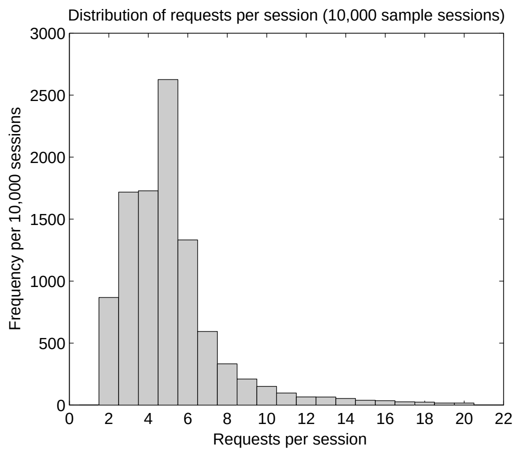


Figure 3.5: Distribution of the number of requests per session for the Gold, Silver, and Bronze services. The distribution peaks at 5 requests per session, and follows a long-tailed Pareto distribution up to 20 requests per session.

for a response from the system before initiating the next request. State information is maintained between multiple requests belonging to one user session. We use a long-tailed Pareto distribution shown in Fig. 3.5, typical of many web-based workloads [12, 32], to model a variable number of requests per session. The distribution has a mean of 5 requests per session, spanning a range from a minimum of 2 requests per session to a maximum of 20.

- **Time-varying requests**. The number of new sessions arriving at each time instance varies. We use time-varying workload traces from the Soccer World Cup 1998 Web site [4], re-interpreting the request arrivals at each data point as session arrivals. As shown by the example trace in Fig. 3.6, session arrivals exhibit time-of-day variations typical of many enterprise workloads, and the number of arrivals changes quite significantly within a very short time period [4, 41]. Each data point in Fig. 3.6 represents the number of new session arrivals during a 30-second time interval.

- **Time-varying transaction mix**. The transaction mixes for all three applications will vary with time, from an all browsing mix (involving database reads) to an all buy-

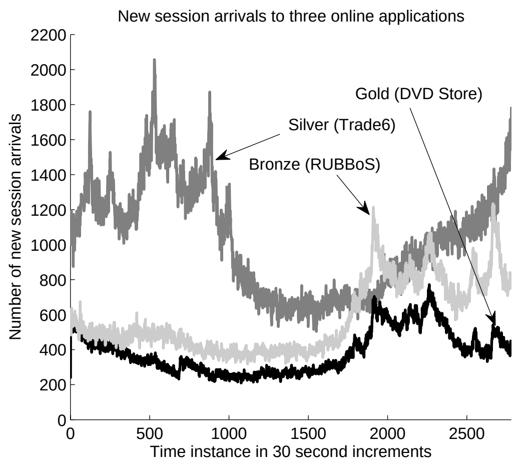


Figure 3.6: New session arrivals to the Gold, Silver, and Bronze applications, plotted at 30-second intervals.

ing mix (involving database writes). Typically, browse requests will have shorter response times than buy requests, since buy requests involve a phased process to commit the transaction. The controller is required to adapt to the corresponding variations in response time. For example, the controller will increase the workload intensity on VMs during an all-browse period. We dynamically change the transaction mix of the workload at several instances during run time.

- **Known user think time**. The user waits a known, fixed amount of time before issuing a new request after receiving a reply to the previous request. The user's think time before each request, following a reply, assumes an average of 4 seconds for all sessions, and over all applications.

- **Varying request inter-arrival time**. The system reply time will vary with each request. Therefore, the inter-arrival time of the workload will too vary, as the user waits for a reply from the system before sending the next request. The maximum inter-arrival time is bounded by the user's think time, plus the response time from the application. Each VM will produce naturally occurring variations in response time, as well as those variations due to the resource assignment (CPU share) of the VM.

Finally, the workload generated for our system should never exceed the system's capacity to process that workload. Given the system setup in Fig. 3.3, we experimentally determine the worst-case workload intensity in terms of Gold, Silver, and Bronze request-arrivals that can be handled by our system, and thus, learn the maximum processing rate and establish an admission policy to cap the maximum arrivals. This ensures that the system can meet the SLAs shown in Fig. 3.2, given an initial cluster configuration under the peak workload. An admission policy also prevents against monetary losses and ensures a fair comparison between the controlled and uncontrolled systems in our results.

We perform the following analysis to set the maximum arrival rates for the Gold, Silver and Bronze services. First, we add up the total processing capacity of the system, measured in units of CPU frequency. The five host machines at the application tier provide a cumulative processing rate of about 54 GHz to incoming requests. We choose a policy wherein the three services equally share the available capacity, that is, as part of the off-line planning process, we allocate 33% of the overall capacity, about 18 GHz, to each service[2]. Now, we must determine the worst-case arrival rate that can be processed by a service without SLA violations, and we do this by profiling the corresponding applications. For example, our experiments indicate that a 6 GHz VM hosting Trade6 (the Silver service), when provided with a 0/100 mix of browse/buy requests, can process about 30 requests/second before response times begin to exceed the SLA. Since the Silver service is allocated 18 GHz on our system, the maximum arrival rate tolerated is $3 \times 30 = 90$ requests per second. Similar calculations for the Gold and Bronze services show that the maximum arrival rates tolerated are 54 and 60 requests per second, respectively. Establishing arrival rate caps ensures that the uncontrolled system, with its fixed allocation of resources, will always have enough processing capacity for the incoming workload. This also ensures a fair comparison

[2]This policy is not unique and any good capacity-planning process will suffice.

between the controlled and uncontrolled system in our results.

# 4. Problem Formulation and Controller Design

This chapter describes how we formulate the resource provisioning problem, introduced in Chapter 2, within the LLC framework. We approach the problem as one of sequential optimization under uncertainty, and explicitly encode a notion of risk into the optimization problem. The discussion includes details of the controller architecture and implementation. The material presented in this chapter was previously published in [26–30].

## 4.1 Introduction

The online optimization framework developed in this thesis combines power and performance management in a virtualized computing environment serving session-based workloads. The framework allows system administrators to satisfy SLA goals while achieving higher server utilization and energy efficiency by dynamically provisioning VMs, consolidating the workload, and turning servers on and off as needed.

The design of our control framework focuses on the following key practical issues.

- We show how simulation-based learning can be used to construct approximate dynamical models of the software components under control. These models capture the non-linear relationship between the response time achieved by a VM, and the CPU and memory share provided to it.

- The LLC formulation models the cost of control, i.e., the switching costs associated with turning machines on or off. For example, profits may be lost while waiting for a VM and its host machine to be turned on, which is usually four to five minutes. Other switching costs include the power consumed while a machine is being powered up or down, and not performing any useful work.

- Excessive switching of host machines and VMs may occur in an uncertain operating environment where the incoming workload is highly variable. This may increase the number of SLA violations, especially in the presence of switching costs. Thus, each provisioning decision made by the controller is risky and we use preference functions to explicitly encode the notion of risk in the LLC problem.

- User-defined policies can be specified as operating constraints to the controller. Since excessive power cycling of a server can reduce its reliability, we show how system administrators can control the power cycling by specifying a policy to leave servers powered on for a minimum period of time.

The prior work reported in [25, 55, 56, 61] is different from that developed in this thesis in that we apply a proactive control technique that encodes the risk involved in making provisioning decisions in a dynamic operating environment and accounts for the corresponding switching costs. In [25], an online method selects a VM configuration to execute on a minimum number of host machines. Events such as CPU utilization and memory availability trigger revision of the placement of VMs. The reactive control method accounts for VM migration costs, but not for the time delays and opportunity costs incurred when switching hosts and VMs on/off. No power is saved in [25] by switching off host machines during periods of light workload.

The server consolidation technique described in [56] reduces the energy consumed by a web application hosting environment. The approach assumes a single application hosted on homogeneous servers, and estimates the CPU processing capacity needed to serve the incoming workload. The dynamic VM rescheduling and collocation scheme in [55] considers heterogeneous workloads. The scheduling algorithm assigns a mix of interactive and batch tasks across a cluster of physical hosts and VMs to meet user-specified deadlines. This work uses migration to re-assign VMs to host machines as new tasks arrive. While [55] consolidates VMs, it does not save power by switching off unused host machines, and does not incorporate migration costs into the control problem.

The authors of [61] present a two-level optimization scheme that allocates CPU share to VMs processing two enterprise applications on a single host. Each VM houses a local controller that uses fuzzy-logic models to estimate the CPU share needed to satisfy the current workload intensity, and makes requests to a global controller for this CPU share. The global controller acts as an arbitrator between multiple VMs, seeking to maximize the profit generated by the host machine. The power and performance framework developed in [23] uses dynamic voltage scaling of the CPUs on the operating hosts to demonstrate a $10\%$ savings in power consumption with a small sacrifice in performance.

The problem of reducing power consumption in server clusters has been well-studied in recent work [31,46,47]. Throttling the CPU clock and dynamic voltage scaling are combined with switching entire servers on/off as needed, based on the incoming workload, to reduce power consumption. Two crucial issues, however, must be addressed in the presence of switching costs. First, turning servers off in a dynamic environment is somewhat risky in QoS terms—excessive switching can increase SLA violations. Second, excessive power cycling of a host machine, that is, turning the machine on and off repeatedly, can lead to reduced reliability [18,58]. The risk-aware controller with added policy constraints presented in this thesis is a step towards addressing the concerns of excessive power cycling and reduction of SLA violations.

Before we discuss the development of the controller, we briefly restate the resource provisioning problem for the computing system of interest. Given the system architecture in Fig. 4.1 and the SLA functions in Fig. 3.2, the control objective is to maximize the profit generated by the Gold, Silver, and Bronze services under a time-varying workload by dynamically tuning the following parameters: (1) the number of VMs to provision to each application, $n_i(k)$; (2) the number of hosts on which to collocate the VMs, $N(k)$, including the migration of VMs; (3) the CPU share to be given to each VM, $f_{ij}(k)$; (4) the number

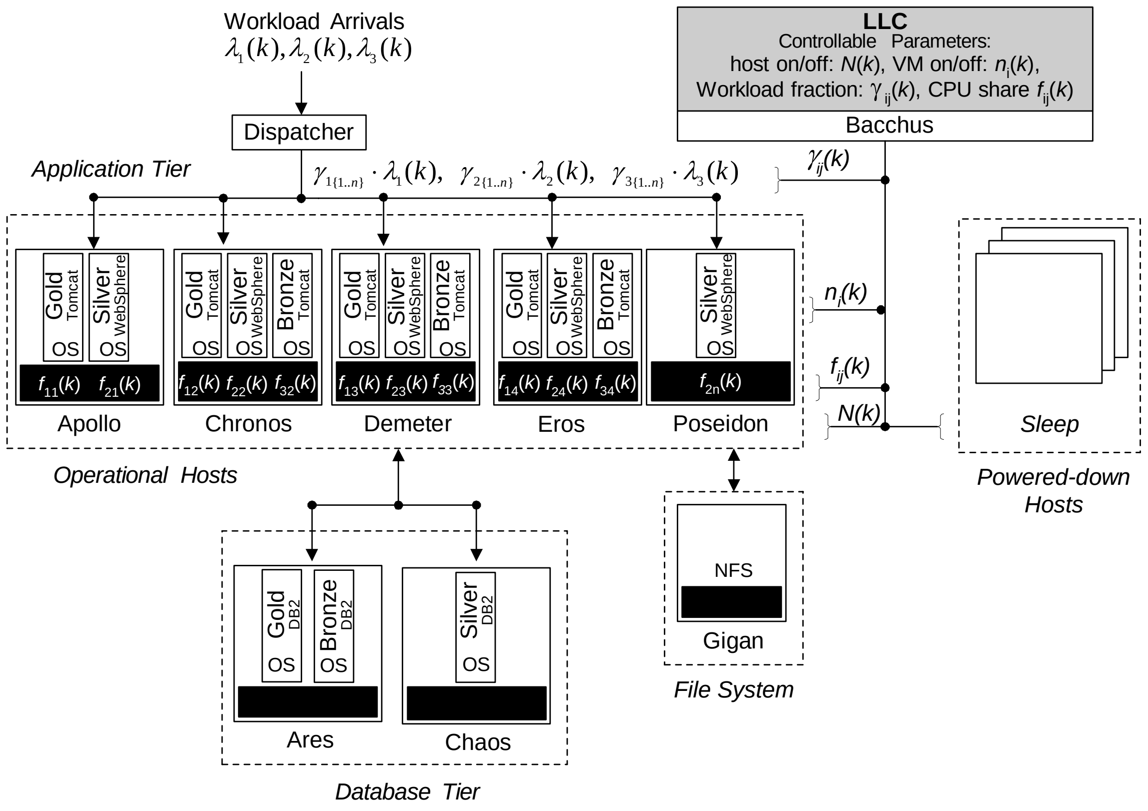

Workload Arrivals
$\lambda_1(k), \lambda_2(k), \lambda_3(k)$
Dispatcher
LLC
Controllable Parameters:
host on/off: N(k), VM on/off: $n_i(k)$,
Workload fraction: $\gamma_{ij}(k)$, CPU share $f_{ij}(k)$
Bacchus
Application Tier
$\gamma_{1\{1..n\}} \cdot \lambda_1(k)$, $\gamma_{2\{1..n\}} \cdot \lambda_2(k)$, $\gamma_{3\{1..n\}} \cdot \lambda_3(k)$
$\gamma_{ij}(k)$
Gold Tomcat OS
Silver WebSphere OS
Bronze Tomcat OS
$f_{11}(k)$ $f_{21}(k)$
$f_{12}(k)$ $f_{22}(k)$ $f_{32}(k)$
$f_{13}(k)$ $f_{23}(k)$ $f_{33}(k)$
$f_{14}(k)$ $f_{24}(k)$ $f_{34}(k)$
$f_{2n}(k)$
Apollo
Chronos
Demeter
Eros
Poseidon
$n_i(k)$
$f_{ij}(k)$
N(k)
Sleep
Operational Hosts
Powered-down Hosts
Gold DB2 OS
Bronze DB2 OS
Silver DB2 OS
Ares
Chaos
NFS
Gigan
File System
Database Tier

Figure 4.1: The system architecture showing the various control inputs. The controller sets $N(k)$, the number of active hosts, $n_i(k)$, the number of VMs to serve the $i^{th}$ application, and $f_{ij}(k)$ and $\gamma_{ij}(k)$, the CPU share and the fraction of workload to distribute to the $j^{th}$ VM, respectively. A *Sleep* cluster holds powered-off machines.

of servers to power up or down; and (5) the distribution of workload, $\gamma_{ij}(k)$.

This chapter breaks down the controller design into three major topics. First, we describe how dynamical models of the computing cluster are developed. Next, we formulate the profit maximization problem, and lastly, we present our implementation of the control framework to solve the profit maximization problem.

## 4.2 Modeling the System Dynamics

We define a *virtual computing cluster* as a group of VMs distributed across one or more physical machines, cooperating to host one online service. The dynamics of a virtual cluster for the Gold, Silver, and Bronze applications are described by the discrete-time state-space equation[1]

$$x_i(k+1) = \phi\big(x_i(k), u_i(k), \omega_i(k)\big) \tag{4.1}$$

where $x_i(k)$ is the state of the cluster, $\omega_i(k)$ denotes the environment input, and $u_i(k)$ is the control input. The behavioral model $\phi$ captures the relationship between the system state, the control inputs that adjust the state parameters, and the environment input.

The operating state of the $i^{th}$ virtual cluster is denoted as $x_i(k) = \big(r_i(k), q_i(k)\big)$ where $r_i(k)$ is the average response time achieved by the cluster and $q_i(k)$ is the number of queued requests. The control input to the $i^{th}$ virtual cluster is denoted as

$$u_i(k) = \big(N(k), n_i(k), \{f_{ij}(k)\}, \{\gamma_{ij}(k)\}\big) \tag{4.2}$$

where $N(k)$ is the system-wide control variable indicating the number of active host machines, $n_i(k)$ is the number of VMs for the $i^{th}$ service, $f_{ij}(k)$ is the CPU share, and $\gamma_{ij}(k)$ is workload fraction directed to the $j^{th}$ virtual machine. The environment input $\omega_i(k) = \{\lambda_i(k), m_i(k)\}$ includes the workload arrival rate $\lambda_i(k)$ and the transaction mix $m_i(k)$ to the $i^{th}$ virtual cluster.

An estimate for the environment input $\lambda_i$ for each step along the prediction horizon includes requests generated by existing sessions in the system plus an estimated number of requests for new sessions. The time-varying nature of the workload makes it impossible to assume an *a priori* distribution. Therefore, in our work, a Kalman filter is designed that uses a Holt-Winter forecasting model to make workload predictions [15]. To estimate the environment input $\lambda$ at time $k$, denoted by $\hat{\lambda}(k)$, the Holt-Winter forecasting model uses an exponentially weighted moving average (EWMA) model to calculate the mean $u$, and a

[1]We use the subscript $i$ to denote the $i^{th}$ service class; $i \in \{1, 2, 3\}$ denotes the Gold, Silver, and Bronze services, respectively.

slope component $b$ to capture the trend in the time-series data [15].

$$\hat{\lambda}(k) = u(k-1) + b(k-1) \tag{4.3}$$

Successive estimates of the environment input for time $k+1$ and beyond are obtained simply by extending the time series further into the future with respect to the slope. The mean $u$ at time $k$ is updated as

$$u(k) = \alpha_0 \lambda(k) + (1-\alpha_0)(u(k-1) + b(k-1)) + \Upsilon \tag{4.4}$$

where $\alpha_0$ is a smoothing constant and $\Upsilon$ is a white-noise disturbance. The slope $b$ at time $k$ is updated similarly as

$$b(k) = \alpha_1(u(k) - u(k-1)) + (1-\alpha_1)b(k-1) + \xi \tag{4.5}$$

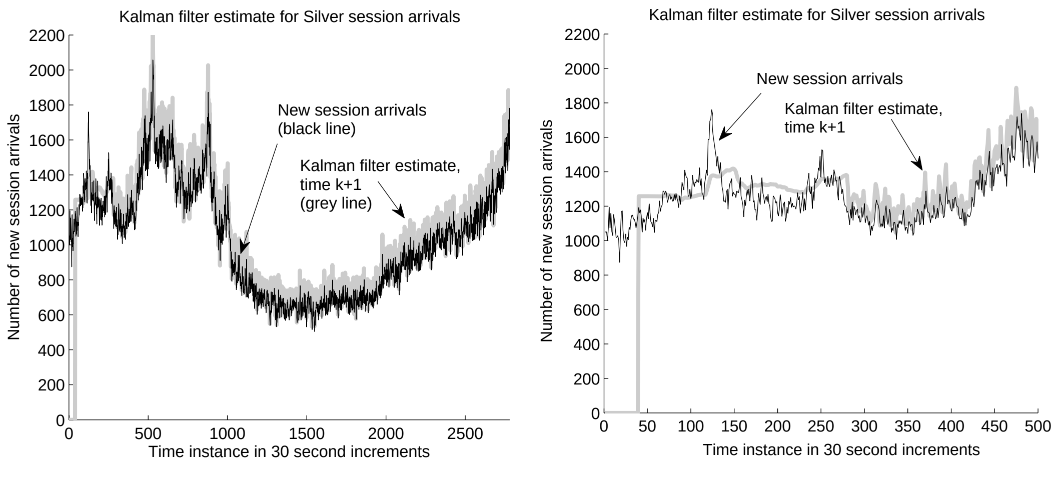
where $\alpha_1$ is a smoothing constant and $\xi$, again, is a white-noise disturbance. The Kalman filter can be trained using representative data to obtain values for $\alpha_0$ and $\alpha_1$ such that the sum of squared errors is minimized for a one-step-ahead forecast as

$$S(\alpha) = \min_{\alpha_0,\alpha_1} \sum (\lambda(k) - \hat{\lambda}(k)) \tag{4.6}$$

where $\lambda$ and $\hat{\lambda}$ are the actual and predicted workload values at time $k$, respectively.

Fig. 4.2(a) shows a sample workload for the Silver application and the one-step-ahead estimate provided by the Kalman filter. The average absolute error for the estimate is about 5%. Fig. 4.2(b) shows an insert of Fig. 4.2(a) for the first 500 sampling increments, during which the Kalman filter is trained for the first 40 samples.

Since the actual values for the environment input cannot be measured until the next



(a) (b)

Figure 4.2: (a) A sample workload for the Silver application and the corresponding predictions provided by the Kalman filter. The average absolute prediction error for the one-step-ahead estimate is about 5%. (b) An insert of Fig. (a) showing the training period for the filter, followed by a period of successive improvements in the workload predictions.

sampling instant, the system state for virtual cluster $i$ at time $k$ is estimated as

$$\hat{x}_i(k+1) = \phi\big(x_i(k), u_i(k), \hat{\omega}_i(k)\big) \tag{4.7}$$

where $\phi$ is developed as difference equations for each virtual cluster $i$ as follows.

$$\hat{x}_i(k+1) = \begin{pmatrix} \hat{q}_i(k) \\ \hat{r}_i(k) \end{pmatrix} \tag{4.8}$$

$$\hat{q}_i(k) = \sum_{j=1}^{n_i(k)} \max\{\big(\gamma_{ij}(k) \cdot \hat{\lambda}_i(k) - \mu_{ij}(k)\big) \cdot T_s, 0\} \tag{4.9}$$

$$\hat{r}_i(k) = g\big(\mu_i(k), \hat{\lambda}_i(k)\big) \tag{4.10}$$

| Symbol | Description |
|---|---|
| | **Observable variables** |
| $\lambda_i(k)$ | Request arrival rate to the $i^{th}$ virtual cluster |
| $m_i(k)$ | Transaction mix of requests to the $i^{th}$ virtual cluster |
| | **Estimated variables** |
| $q_i(k)$ | Queue length of the $i^{th}$ virtual cluster |
| $\mu_i(k)$ | Processing rate of the $i^{th}$ virtual cluster |
| $r_i(k)$ | Average response time of the $i^{th}$ virtual cluster |
| | **Control variables** |
| $n_i(k)$ | Size of the $i^{th}$ virtual cluster |
| $N(k)$ | Number of operational host machines |
| $f_{ij}(k)$ | CPU share of the $j^{th}$ VM in the $i^{th}$ virtual cluster |
| $\gamma_{ij}(k)$ | Fraction of the $i^{th}$ workload to the $j^{th}$ VM |

Figure 4.3: Explanation of the symbols used in Equations (4.9)-(4.12) to describe the system dynamics.

where $\hat{\lambda}_i(k)$ and $\mu_i(k)$ are developed as

$$\hat{\lambda}_i(k) = \hat{\lambda}_i^K(k) + \frac{q_{ij}(k)}{T_s} \tag{4.11}$$

$$\mu_i(k) = \sum_{j=1}^{n_i(k)} \big(\mu_{ij}(k)\big), \quad \mu_{ij}(k) = p(f_{ij}(k), \hat{m}_i(k)) \tag{4.12}$$

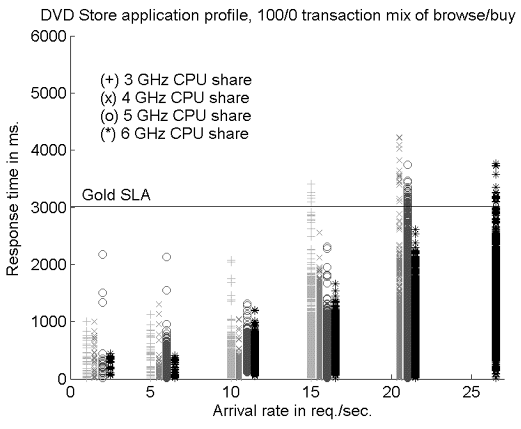
Equations (4.9)-(4.12) capture the system dynamics over $T_s$, the controller sampling time. The estimated queue length $\hat{q}_i(k) \geq 0$ is obtained using the the estimated incoming workload $\hat{\lambda}_i(k)$ dispatched to the cluster and the processing rate $\mu_i(k)$. The estimated workload $\hat{\lambda}_i(k)$ to be processed by the virtual cluster is now given by the fraction $\gamma_{ij}(k)$ of the Kalman estimate $\hat{\lambda}_i^K(k)$ given to the $j^{th}$ VM plus the current queue length (converted to a rate value) of the VM. The estimated transaction mix $\hat{m}_i(k)$ is simply the transaction mix observed during the previous sampling interval. The symbols used in the above equations are summarized in Fig. 4.3.

The processing rate $\mu_i(k)$ of a virtual cluster in (4.12) depends on the number of VMs and the CPU share given to each VM. Each VM is given a share of the host machine's

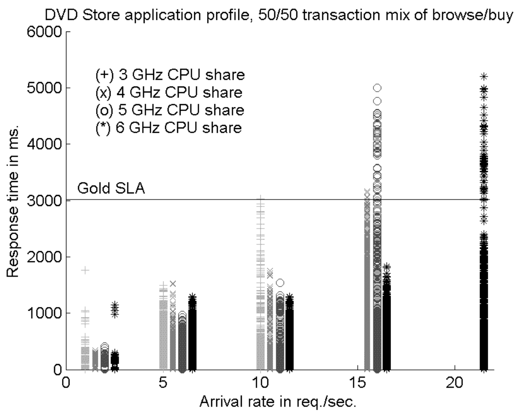
CPU, memory, and network I/O, and (4.12) uses the function $p(\cdot)$ to map the CPU share of the $j^{th}$ VM in the cluster to a corresponding processing rate. This function is stored as a lookup table, indexed by the CPU share $f_{ij}(k)$ and the estimated transaction mix $\hat{m}_i(k)$. The estimated response time $\hat{r}_i(k)$, output by the function $g(\cdot)$ in (4.10), maps the request processing and arrival rates for a given transaction mix to a response time.

We obtain the functions $p(\cdot)$ and $g(\cdot)$ by *simulation-based learning*, that is, by taking the system offline and profiling its behavior under a range of workload conditions. To do this, we first partition the system into three, assigning one third of the available capacity to each application as per the capacity planning process described in Section 3.4. Since the total processing capacity of the system is about 54 GHz, one third corresponds to 18 GHz, or three 6 GHz VMs hosting an application at one time. We give two of the three VMs a maximum CPU share, and load them with a constant workload intensity. We then profile the third VM by setting its CPU share and measuring its response times under an increasing workload intensity. Keeping two machines busy during the test ensures that we stress both the application and the database tier at the same time. The testing is repeated for transaction mixes having 0/100, 50/50, and 100/0 ratios of browse/buy requests within each session.

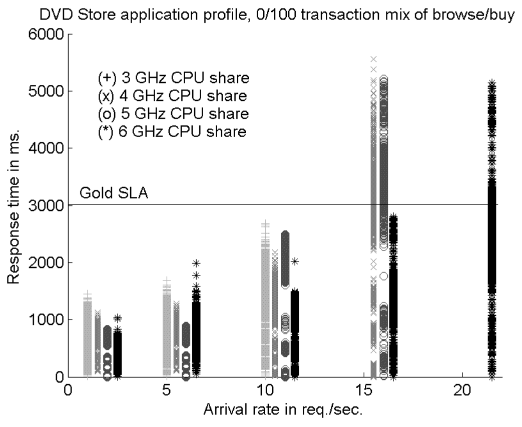
Figs. 4.4 through 4.6 show the response times achieved by a VM under different test scenarios. The SLA goal is indicated by the horizontal dotted line; our objective is to keep 99.9% of the response times within the SLA. The arrival rate at which response times begin to violate the SLA goal indicates the VM's maximum processing rate for that application $p(\cdot)$, given that CPU and memory share. For example, Fig. 4.5 shows that a 6 GHz VM can process approximately 30 Silver requests per second under a 50/50 browse/buy transaction mix before queueing instability occurs. If the VM's CPU share is further constrained, say to 3 GHz, its maximum processing rate decreases and SLA violations occur earlier, at about 20 requests per second.



(a)



(b)



(c)

Figure 4.4: Response times achieved by DVD Store, as a function of a VM's CPU share, arrival rate, and workload mix. Each VM is allocated 770 MB of memory. The shortest response times are achieved with an all browsing mix.

We obtain $g(\cdot)$ in similar fashion. Consider, for example, Fig. 4.5(c) when the arrival rate is 20 requests per second with 0/100 transaction mix. We assign a CPU share of 3 GHz to the VM under test, holding the two other VMs in the virtual cluster at 6 GHz and stressing them (and the database tier) with the maximum arrival rate that can be tolerated without SLA violations (about 30 requests per second). Now, we start providing 20 requests per second to the VM under test, and measure the response time achieved. The experimental data shows us that 99.9% of requests satisfy a response time of about 1800 ms. The function

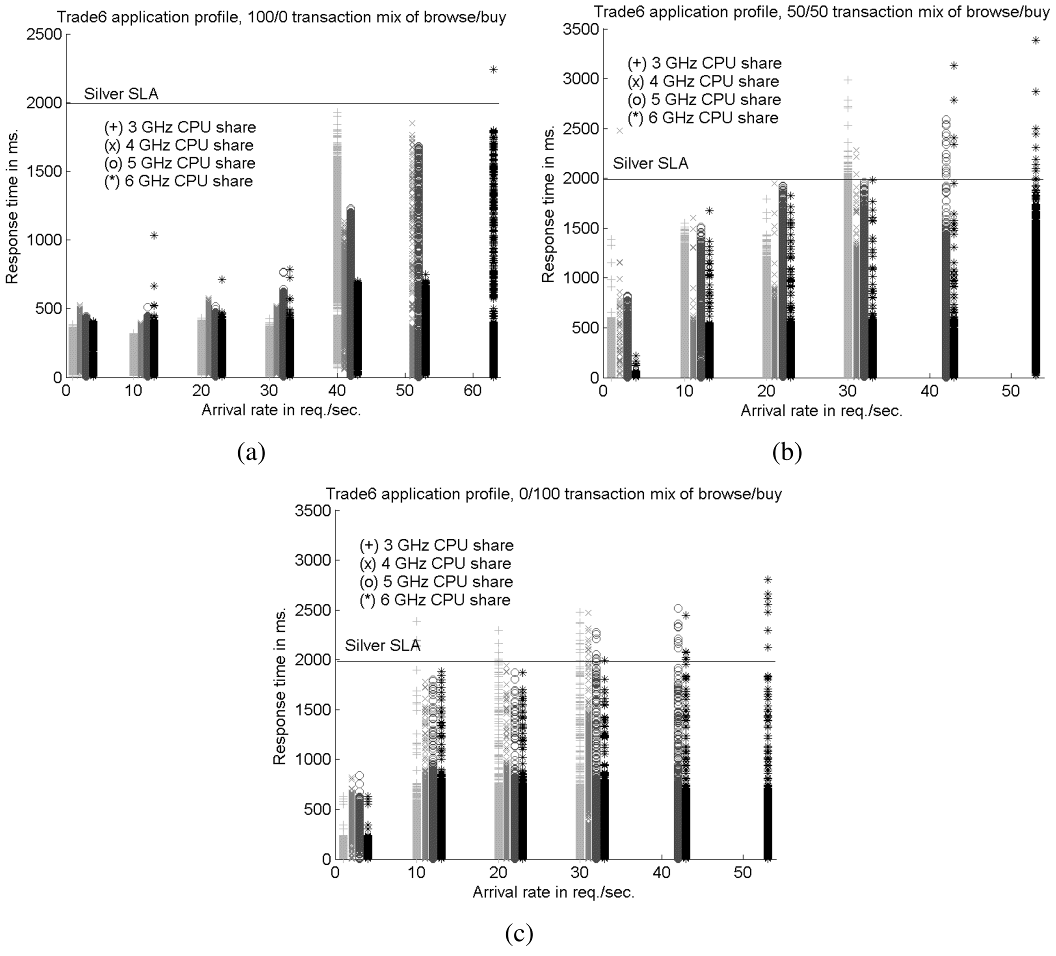
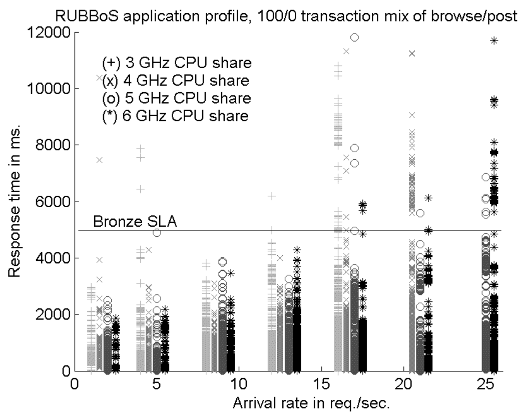


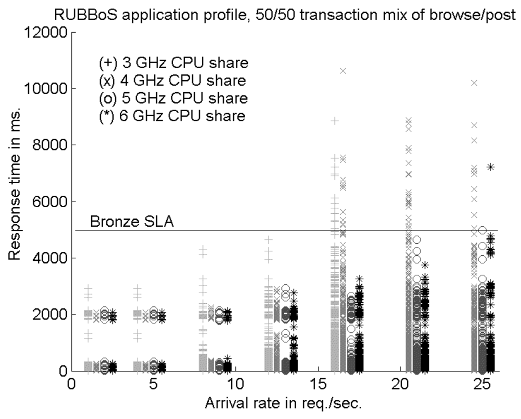
Figure 4.5: Response times achieved by Trade6, as a function of a VM's CPU share, arrival rate, and workload mix. The shortest response times are achieved with an all browsing mix.

$g(\cdot)$ will then output a response time of 1800 ms for a VM given this CPU share, workload intensity, and transaction mix.

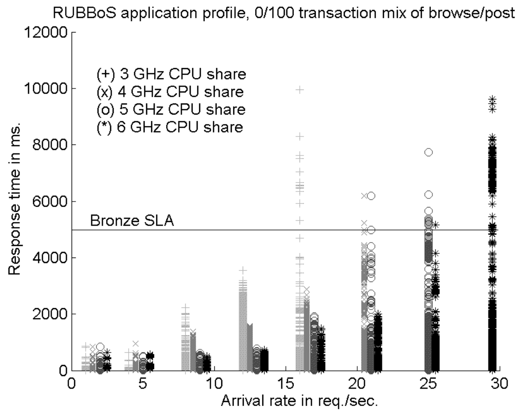
For the DVD Store and Trade6 applications, workloads consisting of more browse requests (database reads) than buy requests (database writes) impose less stress on the system, and these services tolerate higher arrival rates when the workload mix is mostly browse requests. RUBBoS, however, has a large database size, as discussed in Section 3.2, that causes browsing for messages to incur higher response times than posting messages. Therefore, a workload of mostly browse requests will cause the maximum tolerated arrival rate



(a)



(b)

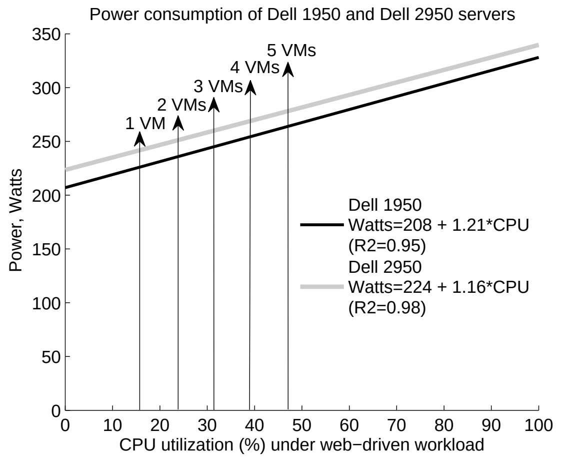


(c)

Figure 4.6: Response times achieved by RUBBoS, as a function of a VM's CPU share, arrival rate, and workload mix. The longest response times are incurred by an all browsing mix, due to the large size of the database.

to decrease. For transaction mixes that lie between the browe/buy mixes shown in Figs. 4.4 through 4.6, we use linear interpolation to estimate the maximum arrival rates tolerated by the various VMs, given different CPU shares. This interpolation enables the controller to adapt to a variable transaction mix at run-time.

The power consumption of the host machine is also profiled off-line by placing it in the different operating states shown in Fig. 4.7(a). Using a clamp-style ammeter, we measured the current drawn by the servers as we instantiated VMs, and loaded each one with an increasing workload intensity before booting the next one. We then multiplied the measured



(a)

| Host State | Power (Watts) Dell 1950 | Power (Watts) Dell 2950 |
|---|---|---|
| Standby | 18 | 20 |
| Shutdown | 213 | 228 |
| Boot | 288 | 299 |
| 1 VM | 226 | 299 |
| 2 VMs | 237 | 250 |
| 3 VMs | 246 | 260 |
| 4 VMs | 255 | 269 |

(b)

Figure 4.7: (a) The power consumed by two models of Dell servers when loaded with VMs hosting an application server; the line is fit from experimentally collected data. (b) The power model used by the controller.

current by the rated wall-supply voltage to compute the power consumption in Watts.

We also computed the power consumed by our servers when booting up, powering down, and in a standby state, during which only the network card is powered on; the power consumption during these two states is included in Fig. 4.7(b). The Dell PowerEdge 1950 and 2950 servers consume 218 and 228 Watts, respectively, when booting up, and 213 and 228 Watts, respectively, when powering down. The same machines consume 18 and 20 Watts in a standby state. To determine the cost of operating the host machine during each controller sampling interval $T_s$, the server's power consumption is multiplied by a dollar cost per kilo-Watt hour over $T_s$.

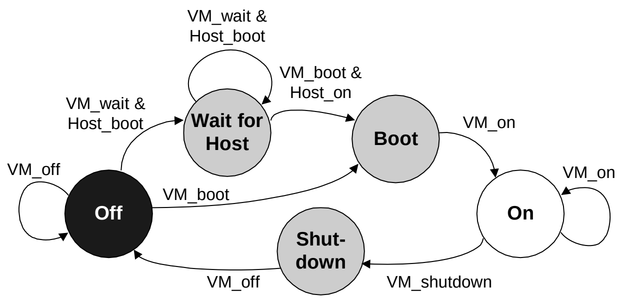
Inspecting the data in Fig. 4.7, we make the following observations regarding the power consumption of the Dell servers.

- An idle machine consumes 70% or more of the power consumed by a machine running at full CPU utilization. Therefore, to achieve maximum power savings on a lightly loaded machine, it is best to redirect the incoming workload and power down the machine. Other power-saving techniques, such as dynamic voltage scaling (DVS), that allow processors to vary their supply voltages and operating frequencies from within a limited set of values [21], have been shown to achieve only about a 10-20% reduction in energy consumption [23, 47].

- The intensity of the workload directed at the VMs does not affect the power consumption and CPU utilization of the host machine. The host machine draws the same amount of current regardless of the arrival rate experienced by the VMs; only the number of VMs running on the host machine affects its power consumption.

- The power consumed by a server is simply a function of the number of VMs instantiated on it at any time instance. Fig. 4.7(b) shows the simplified power model used by the controller, derived from Fig. 4.7, which shows power consumption as a function of the number of VMs running on the Dell 1950 and 2950 servers.

The vector $u_i(k)$ to be decided by the controller at sampling time $k$ for each virtual cluster includes $n_i(k) \in Z^+$, the number of VMs to provision, $f_{ij}(k) \in \{3, 4, 5, 6\}$ GHz, the CPU share, $\gamma_{ij}(k) \in \Re$, the workload fraction to give to the $j^{th}$ VM of the cluster, $N(k) \in Z^+$, the number of active hosts, and a mapping of VMs to host machines. The size of the virtual cluster $n_i(k)$ may be modified by instantiating and shutting down VMs, and by changing the service that the VM provides to the client (e.g. from Gold to Silver). Similarly, the number of operating hosts $N(k)$ may be modified by powering up and shutting down



(a)

| Signal | Description |
|---|---|
| VM_wait | Schedule a VM for instantiation; host is not ready |
| VM_boot | Instantiate a VM; host is ready |
| VM_on | VM is instantiated and ready for the workload |
| VM_shutdown | VM is being shut down and resources de-allocated |
| VM_off | VM is turned off |
| Host_boot | Host machine is booting up |
| Host_on | Host machine is powered on |

(b)

Figure 4.8: (a) The finite state machine corresponding to the various operating states of a virtual machine. A VM requires two state transitions to power down, and up to four state transitions to turn on. (b) Descriptions of the state-transition signals.

machines, aided by VM migration that enables the consolidation of VMs onto fewer host machines.

Control delays present additional challenges that must be modeled within the control problem. The CPU share $f_{ij}(k)$ and workload distribution $\gamma_{ij}(k)$ to VMs can be actuated immediately; that is, there is zero time delay between deciding the control input and realizing its effect on the system. Realizing the controller-specified number of host machines $N(k)$, however, incurs a delay during the time a server is powering up or shutting down. For example, about three minutes are required to power up a host machine. Similarly, realizing the number of VMs $n_i(k)$ incurs a delay to power up or shut down virtual machines, and the duration of this delay depends on the state of the host machine to which a VM is assigned. Migration of VMs using VMotion and changing applications on a live VM both incur small delays of 30 seconds each.

Given the control delays, the challenge for the controller is to maintain state information for each host machine and VM at every step of the prediction horizon. This is done using the VM state-transition diagram shown in Fig. 4.8(a). If the controller decides to turn on a VM, it must first evaluate the state of the host machine to which the VM is assigned. If the host is turned off, the VM transitions from the *Off* state to the *VM_wait* state, cycling for another time step while the host machine powers up. Finally, the VM transitions to the *VM_boot* state, and is ready to process workload by the fourth transition to the *On* state. Each state is then evaluated by the controller for its profitability, that is, for power consumption costs and revenue generation.

## 4.3 The Profit Maximization Problem

Once we have developed the relevant system models, we can now formulate the profit maximization problem to be solved by the controller for our virtualized computing environment. If $x_i(k)$ denotes the operating state of the $i^{th}$ virtual cluster and

$$u_i(k) = \big(N(k), n_i(k), \{f_{ij}(k)\}, \{\gamma_{ij}(k)\}\big) \tag{4.13}$$

is the decision vector, at time $k$, the controller estimates the profit generated from time $k+1$ through time $k+h$ as

$$P\big(x(k), u(k)\big) = \sum_{l=k+1}^{k+h} \sum_{i=1}^{3} H_i\big(r_i(l)\big) - O\big(u(l)\big) - S(\Delta N(l), \Delta n(l)) \tag{4.14}$$

where the revenue $H_i(r_i(k))$ is obtained from the corresponding SLA function $H_i$ that classifies the response time achieved per transaction into one of two categories, "satisfies SLA" or "violates SLA", and maps it to a reward or refund, respectively. The power-consumption cost incurred in operating $N(k)$ machines is given by $O(k) = \sum_{j=1}^{N(k)} \big(O\big(N_j(k)\big)\big)$ that sums the power-consumption costs incurred by the host machines $O(N_j)$, which is determined by the number of VMs that are assigned to the host machine during a given time interval.

The switching cost incurred by the system due to provisioning decisions is denoted as $S(\Delta N(k), \Delta n(k))$. This accounts for the transient power-consumption costs incurred when powering up/down VMs and their hosts, as well as for the opportunity cost that accumulates during the time a server is being turned on or powered down, performing no useful service. The switching cost also includes the dollar cost incurred when migrating a VM. Experiments indicate that migrating a VM takes 30 seconds, and that during this period, approximately $10\%$ of the requests directed at this VM violate their SLA. So, the cost of migrating a VM is simply the refund paid for $10\%$ of the estimated number of requests arriving during that 30-second interval.

Due to the above-mentioned costs, excessive switching of hosts or VMs, caused by workload variability may actually increase SLA violations and reduce profits. Therefore, we convert the profit-generation function in (4.14) to a corresponding utility function that quantifies a controller's preference between different provisioning decisions in a risky environment. Using utility functions to aid decision making under uncertainty has been studied in the context of investment and portfolio management [7].

To perform risk-aware control, we augment the estimated environment input $\hat{\lambda}(k)$ with an uncertainty band $\hat{\lambda}(k) \pm \varepsilon(k)$, in which $\varepsilon(k)$ denotes the past observed error between the actual and forecasted arrival rates, averaged over a window. For each control input, the next state equation in (4.7) must now consider three possible arrival-rate estimates, $\hat{\lambda}(k) - \varepsilon(k)$, $\hat{\lambda}(k)$, and, $\hat{\lambda}(k) + \varepsilon(k)$ to form a set of possible future states $\mathbf{X}(k)$ that the system may enter. Given $\mathbf{X}(k)$, we obtain the corresponding set of profits generated by these states as

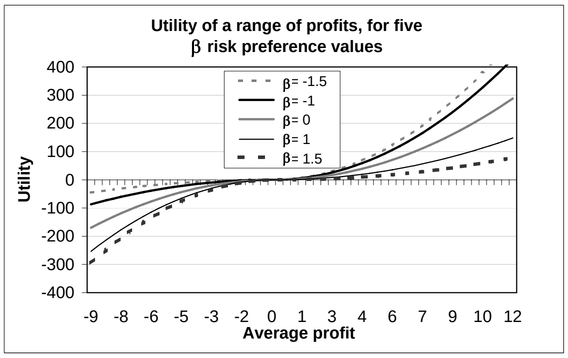
$\mathbf{P}\big(\mathbf{X}(k), u(k)\big)$ and define the quadratic utility function

$$U\big(\mathbf{P}(\cdot)\big) = A \cdot \bar{u}\big(\mathbf{P}(\cdot)\big) - \beta \cdot \Big(\nu\big(\mathbf{P}(\cdot)\big) + \bar{u}\big(\mathbf{P}(\cdot)\big)^2\Big) \tag{4.15}$$

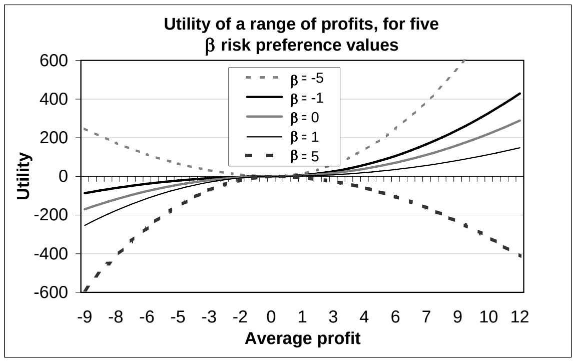
where $A > 2 \cdot \big|\bar{u}\big(\mathbf{P}(\cdot)\big)\big|$ is a constant, $\bar{u}\big(\mathbf{P}(\cdot)\big)$ is the algebraic mean of the estimated profits, $\bar{u}\big(\mathbf{P}(\cdot)\big)$ is the corresponding variance, and $\beta \in \Re$ is a risk preference factor that can be tuned by the data-center operator to achieve the desired controller behavior, from being risk averse ($\beta > 0$), to risk neutral ($\beta = 0$), to risk seeking ($\beta < 0$). Given a choice between two operating states with equal mean profits but with different variances, a risk-averse controller will choose to transition to the state having the smaller variance. The magnitude of $\beta$ indicates the degree of risk preference. We choose the mean-variance utility function in (4.15) for its simplicity as well as the fact that $\beta$ can be used to explicitly influence the controller's performance by setting its risk preference.

Fig. 4.9(a) shows the behavior of the utility function in (4.15) for various values of $\beta$ when we have an increasing average profit and the variance is a constant percentage of the profit. As the figure indicates, given two decisions incurring the same variance, the provisioning decision resulting in the higher average profit will be chosen by both the risk-seeking and risk-averse controller. On the other hand, Fig. 4.9(c) shows the behavior of the utility function under the scenario in which we have an identical average profit, but increase the variance associated with that profit. Here, it is the variance that determines when the preferences of risk-seeking and risk-averse utility functions will diverge, producing different control trajectories.

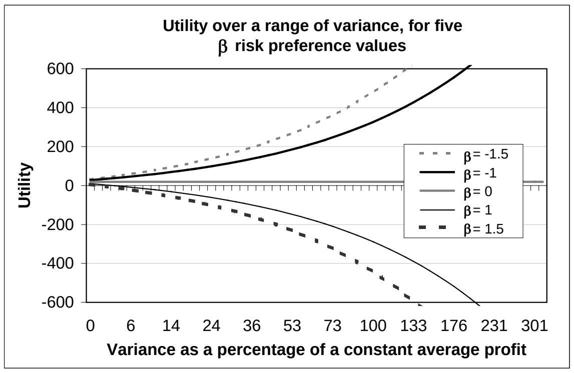
We now show how to obtain a range of $\beta$ values that elicit the desired response from the utility function in (4.15). Further tuning of $\beta$ within this range is then needed to improve controller performance. The range of $\beta$ values must be such that (4.15) assumes a decreasing negative value when the SLA is violated and the estimated profits are less than zero ($\bar{u}\big(\hat{X}_i(l)\big) < 0$), and an increasing positive value when the estimated profits are greater



(a)



(b)



(c)

Figure 4.9: (a) The utility function plotted for five appropriately chosen values of $\beta$, (b) the utility function plotted for three appropriate and two inappropriately chosen values of $\beta$, and (c) the utility function plotted for five values of $\beta$ given the same average profit, but with increasing variance.

than zero ($\bar{u}\big(\hat{X}_i(l)\big) > 0$). Depending on the $\bar{u}$ and $\nu$ values in (4.15), certain values of $\beta$ will be inappropriate. For example, Fig. 4.9(b) plots the utility function for various $\beta$ values when the variance in the generated profits is 25% of the mean. When $\beta = -5$, the utility increases as profits fall below zero, which is not the response we seek from (4.15).

The above example implies that choosing a sensible range of $\beta$ values requires some *a priori* knowledge about the expected variance within a given set of estimated average profits. We assume an upper-bound on the expected variance as $max\big(\nu\big(\hat{X}_i(l)\big)\big) = \bar{u}\big(\hat{X}_i(l)\big)$. Differentiating the utility function with respect to $\bar{u}\big(\hat{X}_i(l)\big)$, we obtain

$$\frac{\partial U}{\partial \bar{u}} = A - \beta\big(1 + 2 \cdot \bar{u}\big(\hat{X}_i(l)\big)\big) \tag{4.16}$$

Solving ( 4.16) for $\beta$ when $\frac{\partial U}{\partial \bar{u}} = 0$ gives us

$$\big|\beta\big| < \frac{A}{1 + 2 \cdot \bar{u}\big(\hat{X}_i(l)\big)} \tag{4.17}$$

We can now choose a value for the coefficient $A$ and obtain $|\beta|$.

Given the utility function in (4.15), we formulate the resource provisioning problem as one of utility maximization.

$$\begin{aligned}
\textbf{Compute:} \quad & \max_{u} \sum_{l=k+1}^{k+h} U\big(\mathbf{P}(\mathbf{X}(l), u(l)), u(l)\big) \\
\textbf{Subject to:} \quad & N(l) \le 5, \quad n_i(l) \ge K_{\texttt{min}}, \quad i = 1, 2, 3 \\
& \sum_{j=1}^{n_i(l)} \gamma_{ij}(l) = 1, \quad i = 1, 2, 3 \quad \texttt{and} \\
& \sum_{i=1}^{3} \sum_{j=1}^{n_i(l)} e_{ijz}(l) \cdot f_{ij}(l) \le F_{\max}^{z}, \quad z = 1 \ldots 5
\end{aligned} \tag{4.18}$$

where $h$ denotes the prediction-horizon length. As an operating constraint, $N(l) \le 5$

ensures that the number of operating servers never exceed the total number of application-tier servers in the testbed, and $n_i(l) \geq K_{\texttt{min}}$ forces the controller to operate at least $K_{\texttt{min}}$ VMs at all times in the cluster to accommodate a sudden spike in request arrivals. In our experiments, $K_{\texttt{min}}$ is set to 1. We also introduce a decision variable $e_{ijz}(l) \in \{0, 1\}$ to indicate whether the $j^{th}$ VM of the $i^{th}$ application is allocated to host $z \in [1, 5]$, and the final constraint ensures that the cumulative CPU share given to the VMs does not exceed $F^z_{\max}$, the maximum capacity available on host $z$.

Dependencies exist between several of the control inputs. For example, the number of virtual machines $n(k)$ needed to satisfy a given workload is dependent upon the CPU share $f(k)$ assigned to each of the machines. If the CPU share to each VM is tightly constrained, we need a larger number of VMs to process the workload. Similarly, the number of VMs needed will decrease as the CPU share increases. Another dependency occurs between the number of operating hosts $N(k)$ and the number of VMs. Because of the heterogeneity among the host machines within our system, the number of VMs a host machine can support and the CPU share to those VMs will vary from host to host. So the characteristics of the host machines in $N(k)$ are an important consideration to ensure that $n(k)$ and $f(k)$ can be satisfied.

We have chosen a centralized controller implementation to decide the control vector for the system at each sampling instance since it can easily accommodate the coupling between control inputs.

## 4.4 The Controller Implementation

Given the system state $x$ at time $k$ and the estimated workload arrival rate $\hat{\lambda}$ for time $k + 1$, the controller first generates a set of valid next states (or system configurations), in terms of the number of hosts and VMs to power on, the allocation of VMs to hosts, and the CPU share of VMs. Each state then is assigned a utility value following (4.15), and the controller selects a state trajectory that maximizes the cumulative utility in (4.18).

To accommodate the actuation delays discussed in Section 4.2, the lookahead horizon for the controller is determined by the maximum time needed to bring a VM online—two control steps to turn on a host, and one control step to turn on a virtual machine. So the minimum lookahead horizon $h$ is three control steps. Once the prediction horizon is fully explored, the controller chooses an optimal trajectory, then actuates the corresponding control input on the system. The entire control process is repeated at time $k + 1$ when updated state information and workload estimates become available.

When control inputs must be chosen from a discrete set, the optimization problem in (4.18) will show an exponential increase in worst-case complexity with an increasing number of control options and longer prediction horizons—the so called "curse of dimensionality." Therefore, to reduce this complexity, we have implemented two heuristic techniques as part of the controller implementation.

*Linear Trajectory Exploration*. To reduce the computational overhead of the controller, we do not explore any additional configurations within the prediction horizon from time steps $k + 2$ to $k + h$, but simply let the configurations generated at time $k + 1$ evolve within the prediction horizon, as shown in Fig. 4.10. Thus, we reduce the computational complexity of (4.18) from exponential in $h$ to linear in $h$. Note that some of the states generated at time $k + 1$ may need to evolve within the horizon before their final impact on system performance can be evaluated. For example, to instantiate a VM on a host that is currently turned off, the controller must account for two time steps to power on the host, and one additional time step to turn on the VM and launch the application.

*Early Elimination of Control Options*. To further reduce the computational overhead of the centralized controller, we use a two-phase approach to estimate the reconfiguration cost which includes the energy costs and the revenue lost during the transition from the current configuration to a new one. An early-elimination scheme is used to pare down the

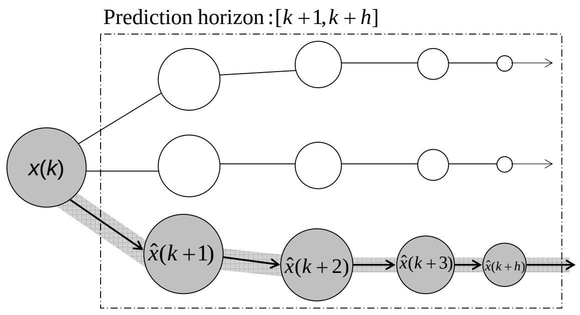


Figure 4.10: The state-space trajectory explored by the controller within a lookahead horizon of length $h$. The shaded trajectory shows the best trajectory chosen by the controller.

roughly 1,300 possible control options into a smaller subset and expedite the optimization algorithm[2]. The basic idea of the two-phase approach is to use a coarse-grain estimator during Phase 1 to rank control options in order of their reconfiguration cost, and then select the top third of the list and pass it to the fine-grain estimator in Phase 2 that will perform the complete optimization routine on each configuration.

During Phase 1, the CPU requirements for each application are estimated based upon the current queue length, the number of new session arrivals, and the transaction mix. Next, the estimator selects the configuration choices that contain a number of VMs to satisfy the CPU needs. The estimator assumes the VMs will be given a maximum CPU share of 6 GHz, thus establishing a lower bound on the number of VMs. Then, the estimator assigns a cost to each configuration based upon its relative distance from the current configuration. This cost is derived from the time delay needed to realize the reconfiguration. For example, if the vector [0,2,1,2,2] represents the number of VMs assigned to each host in the current configuration, and [1,1,1,2,2] is the configuration for time $k+1$ under review, the controller accounts for one host machine being turned on with a delay of 2, and one VM being mi-

[2]The size of the search space can be estimated via $v^N$, where $v = 4$ is the maximum number of VMs per host, and $N = 5$ is the number of controllable host machines.

grated at a cost of 0.5 for the dropped requests[3], for a total cost of 2.5[4]. This process is repeated by increasing the number of VMs from the lower bound to accommodate those choices in which a larger number of VMs are used to satisfy the estimated workload, but each VM is assigned a smaller CPU share. Lastly, the controller ranks all of the control options according to this scheme, and submits the top third with the lowest cost to Phase 2 for fine-grain cost evaluation.

Phase 2 evaluates exactly how to perform the reconfiguration to obtain a more accurate cost estimate. The execution time of Phase 2 is more costly compared to Phase 1, due to the fine-grain analysis of the reconfiguration choices. So, Phase 2 benefits from receiving a limited subset of control options. Phase 2 accounts for switching costs that include: (1) the transient power-consumption costs incurred when powering up/down VMs and host machines, (2) the opportunity cost that accumulates during the time a server is being booted up or powered down, but performs no useful service, and (3) the dollar cost incurred when migrating a VM. Phase 2 also incorporates application re-assignment, that is, when a live VM switches the service it provides to a client, into the VM configuration, as well as the assignment of CPU share to VMs.

Phase 2 applies the switching cost to (4.14) to evaluate each state trajectory in Fig.4.10. The CPU share determines the processing rate and thus, the amount of revenue that the VMs can generate. The mapping of VMs to host machines that occurs in Phase 2 determines the power consumption cost incurred by each configuration. The workload prediction errors are applied to generate a range of estimated profits, then a single utility value is derived for each state according to (4.15). Phase 2 ranks control options for their cumulative utility to the system in (4.18), selects the top value, and applies the corresponding control input to

[3]We assign a small cost to migrating a VM that accounts for the approximately 10% of requests that violate their SLA goal during the 30-second interval of VMotion. We choose a migration cost of 0.5 to be less than the cost of 1 to instantiate a new VM, which incurs a two-minute time delay and opportunity cost.

[4]The controller would not assume that one VM were being turned off, and another one turned on at a cost of a 1 step delay, because it is most efficient to migrate a live VM and switch applications, if so needed.

the system. In the unlikely case that equal utility values result from more than one control option, the control option that is closest to the current configuration is selected.

# 5. Experimental Results

This chapter presents experimental results, analyzing the performance of the LLC framework. To summarize, when managed using a well-tuned controller, the cluster saves, on average, 41% in power-consumption costs over a 24-hour period when compared to a system operating without dynamic control.

This chapter is organized as follows. First, we summarize the main results of our experiments and describe the evaluation criteria. Then, we discuss how the control decisions are actuated within the system, as well as details related to workload generation and workload prediction. Next, we compare the controller performance when operating it in three risk-preference regimes, and show the impact of tuning $\beta$ for risk-aware controller implementations. We also analyze the impact of tuning the prediction-horizon length on the performance of a risk-aware controller. The effect of user-defined policies to reduce server switching activity is evaluated, and lastly, we show that a tuned controller can adapt well to changes in the transaction mix.

## 5.1 Introduction

The LLC framework developed in this thesis is evaluated on its ability to maximize the profit generated by the testbed, shown in Fig. 3.3, over a 24-hour period. According to (4.14), the profit is maximized when both energy consumption and SLA violations are minimized. Therefore, we evaluate the controller using two criteria: (1) the percentage of energy savings over an uncontrolled system, in which all host machines remain in a powered-on state, and (2) the percentage of SLA violations[1].

Fig. 5.1 shows the system and controller parameters used in our experiments. The time

[1]Experiments show that even an adequately provisioned, uncontrolled system will incur a small percentage of SLA violations, typically around 0.01% or more of the total number of processed requests, due to normal variations and the effects of background processes such as Java garbage collection.

| Parameter | Value |
|---|---|
| Cost per KWatt hour | \$ 0.3 |
| Time delay to power on a VM | 1 min. 45 sec |
| Time delay to power off a VM | 45 sec |
| Time delay of a VM migration | 30 sec |
| Time delay to power on a host | 2 min. 55 sec |
| Time delay to power off a host | 1 min. 30 sec |
| Prediction horizon | $\geq$ 3 steps |
| Control sampling period | 2 min. |
| Initial configuration for each service | 3 VMs at 6 GHz each |

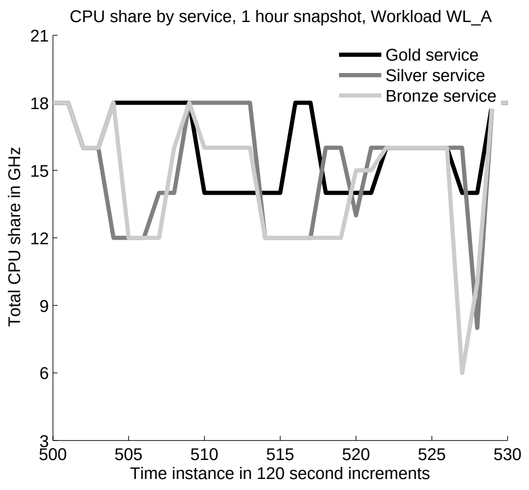
Figure 5.1: The experimental parameters, including the times to power a VM and host machine on and off, control sampling periods, and initial configuration of the VMs.

needed to power up a server is about 2 min. 30 sec., and the time needed to boot a VM and initialize an application on it is about 1 min. 45 sec., and the time to execute the control kernel is about 2 sec. We set the controller sampling time to 2 minutes. Therefore, to evaluate a control decision that involves powering up a host machine and booting a VM on that host, the controller must look ahead at least three sampling intervals. This determines the minimum lookahead horizon length—three steps—for the controller. The time needed to turn off a VM includes the time to shut down the VM and de-allocate its resources on the host machine. Requests belonging to existing sessions on the de-allocated VM are re-routed to other live VMs hosting the same application.

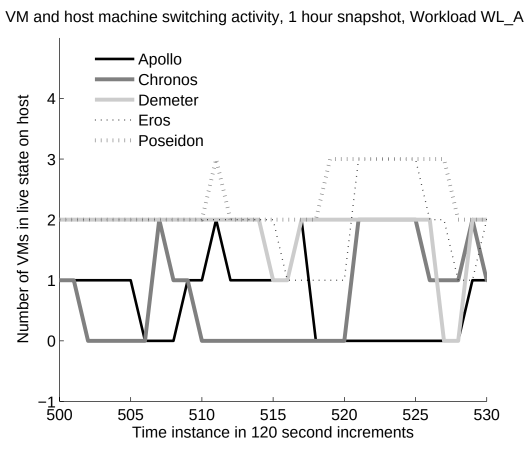
Fig. 5.2 summarizes the results of a risk-averse controller ($\beta = 2$) in terms of energy savings and SLA violations for six enterprise workloads. The energy savings average about 41% when compared to an uncontrolled system over a 24-hour period, and the number of SLA violations, those requests that have spent time in queue, remains very low, about 0.01% in the average case. Fig. 5.3 shows the average CPU share and number of VMs and host machines, respectively, allocated by the controller at each time instance over a 1-hour window for the workload WL_A. In Fig. 5.3(b), a host machine showing zeros VMs indicates that the machine has been powered off. Fig. 5.4 shows the measured response

| | Energy Savings | SLA Violations |
|---|---|---|
| WL_A | 44% | 1,263 (<0.01%) |
| WL_B | 36% | 1,335 (<0.01%) |
| WL_C | 42% | 1,874 (<0.01%) |
| WL_D | 42% | 2,311 (0.01%) |
| WL_E | 42% | 8,818 (0.01%) |
| WL_F | 40% | 5,899 (0.01%) |
| Avg. | 41% | 3,583 (0.01%) |

Figure 5.2: Performance of a risk-averse controller ($\beta = 2$) compared to an uncontrolled system, over a 24-hour period for six different workloads. The average SLA violations are about 0.01% of the total number of requests handled by the system.



(a)

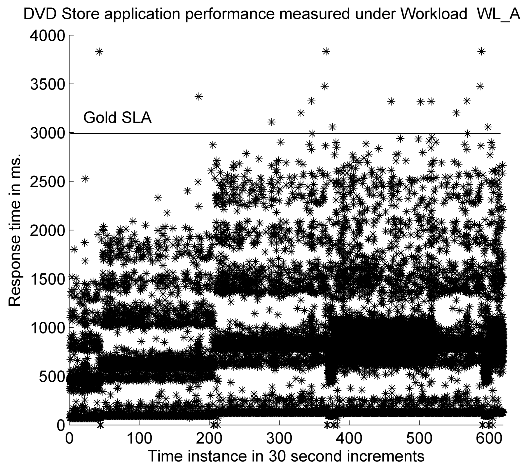


(b)

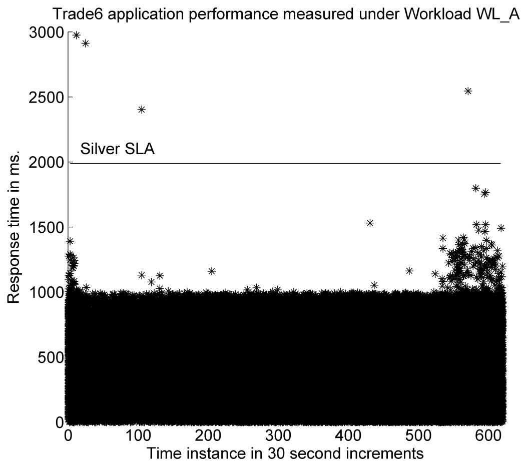
Figure 5.3: (a) Total CPU share by service, and (b) virtual machine and host machine switching activity over a 24-hour period for WL_A. Host machines with zero VMs on them mean that they have been turned off.

times for the Gold, Silver, and Bronze applications under the same workload over a 5-hour window. The measured response times are consistent with those estimated by the system model, that at least 99.99% of response times meet the SLA goals.

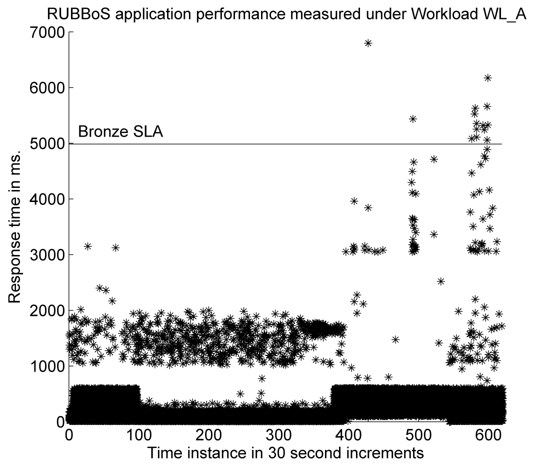
In all our results, an 'uncontrolled system' is one in which the initial configuration of three 6 GHz VMs for each service is unchanged over a 24-hour period. This configuration



(a)



(b)



(c)

Figure 5.4: The measured response times for the Gold (a), Silver 5.4(b), and Bronze (c) applications over a 5-hour window under Workload WL_A.

ensures that each service meets its SLA goal under the worst-case workload arrival rate for a 0/100 browse/buy transaction mix for the DVD Store and Trade6 applications, and a 100/0 transaction mix for the RUBBoS application.

## 5.2 Actuating the Control Decisions

Having introduced the evaluation criteria for the controller and shown a sample set of results, we now discuss details of how the control decisions are actuated. The control kernel is written in Matlab and compiled to a C-language executable for faster execution. A scheduler, written in Java, invokes the controller every 2 minutes (the sampling period). The kernel writes its control directives as a collection of simple text files, which the scheduler reads, and then schedules their actuation at the appropriate time. For example, suppose the control kernel decides to boot a host at time $k$ and turn on a VM at time $k + 2$. If each increment of time $k$ is equal to the control sampling period $T_s = 2$ minutes, then the scheduler will execute the command to turn on a host machine immediately, and execute the command to instantiate a VM after 4 minutes, when has completed its boot routine.

The scheduler actuates the control decisions by communicating with the VirtualCenter and the ESX virtualization layer using the simple object access protocol (SOAP), as well as communicating directly via shell commands to the host machines and VMs. The different control directives are actuated as follows.

- **Migrating a VM**. Migration of VMs is actuated by an API call to the VirtualCenter, the only agent that can authorize VMotion between two ESX servers.

- **Instantiating a VM**. VMs are instantiated by an API call to the ESX virtualization layer on the host machine. A shell script that launches the application of interest is scheduled to execute 1 min. and 45 sec. after the VM has booted. Lastly, the scheduler sends 30 ‘dummy’ workload requests to the VM in an effort to fill the cache with data and ‘warm-up’ the application, improving the response time for the first set of real workload requests.

- **Setting the CPU share for a VM**. The CPU share of a VM is set via an API call to the ESX virtualization layer on the host machine to which the VM is assigned.

- **Setting workload share to a VM**. The workload share to a VM is communicated as a double-precision number within a shared Java object to the workload scheduler. The workload scheduler sends the given proportion of all new incoming sessions to the VM.

- **Shut down a VM**. VMs are shut down by shell commands sent directly to the VM. The scheduler then communicates to the workload generator via a shared Java object a list of the 'backup' VMs to which requests belonging to existing sessions on the de-allocated machine should be re-routed.

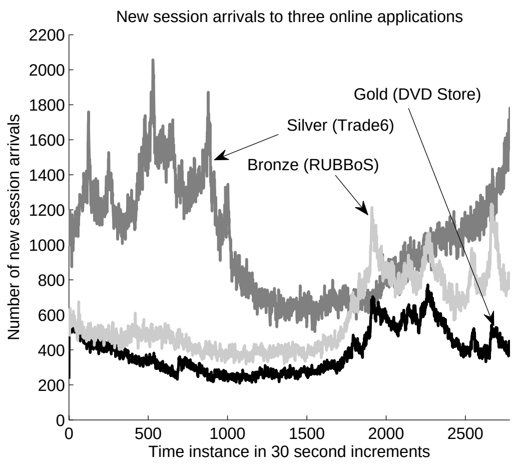
- **Turning on a host**. Host machines are turned on by issuing a wake-on-LAN (WOL) command to the machine's network card.

- **Shutting down a host**. Host machines are turned off by a shell command that cuts the power to the chassis via an Intelligent Platform Management Interface (IPMI) call.

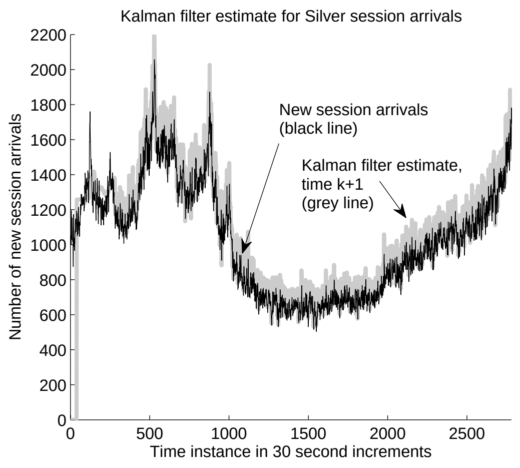
## 5.3 Workload Generation

A closed-loop workload generator sends browse and buy/sell/post requests to the three services. The scheduler described in the previous section triggers the workload generator, also written in Java, to start new sessions within the system every 30 seconds, the time granularity of our World Cup '98 workload traces [4]. As discussed in Section 3.4, the scheduler interprets each data point in Fig. 5.5(a) as the number of new sessions arriving during a 30-second interval. Each new session maintains its own thread within the Java runtime environment.

The session length, for all applications, follows the long-tailed distribution shown in Fig. 5.6 and previously introduced in Chapter 3. The workload generator randomly selects a value from within this distribution and sets the number of requests for each session it



(a)



(b)

Figure 5.5: (a) A sample workload showing new session arrivals to the Gold, Silver, and Bronze applications, plotted at 30-second intervals. (b) A sample workload for the Silver application and the corresponding predictions provided by the Kalman filter. The average absolute error for the one-step-ahead estimate is about 5%.

schedules. The mean session length is 5 requests, the minimum is 2 requests, and the maximum is 20 requests. The time to issue the first request of each new session is uniformly

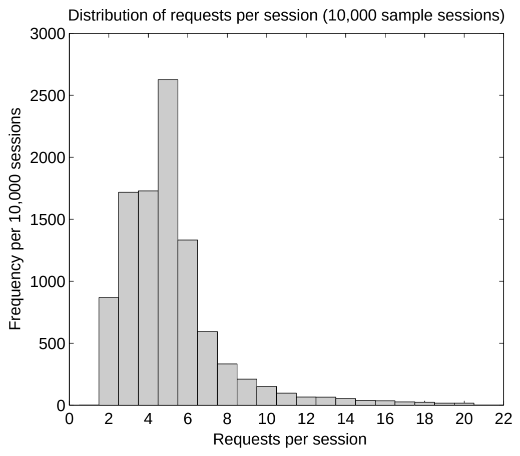


Figure 5.6: Distribution of the number of requests per a sampling of 10,000 sessions for the Gold, Silver, and Bronze services. The mean number of requests is 5 with a minimum of 2 requests per session. The number of requests follows a long-tailed Pareto distribution up to a maximum 20 requests per session.

distributed within the 30-second time period. The remaining requests are issued after a 4-second user think time following the receipt of each response. Because the system response time will vary with the particular request, the inter-arrival time of the intra-session requests will vary as well.

Future session arrivals are predicted by a Kalman filter and sent to the controller. The filter is first trained using a small portion of the workload (the first 40 time steps) and is then used to forecast the remainder of the load during controller execution[2]. Once trained, the filter provides effective estimates—the average absolute error between the predicted and actual values is about 5% for the one-step-ahead estimate, and increases by about 1% for each subsequent estimate within the prediction horizon.

When the controller decides to shut down a virtual machine, new sessions are not sent to the VM, and requests belonging to existing sessions on that VM are re-routed to other VMs hosting the same application. The scheduler instructs the workload generator where to re-route future requests belonging to the existing sessions. Since session lengths follow a long-tail distribution, only a few long sessions will linger, typically on the order of a few minutes, and the burden on the 'backup' VMs will not be significant. Therefore, we use a simple round-robin scheme for re-routing sessions on to the backup VMs. For example, if one VM hosting the Silver application is shut down, and two remain, the workload generator hashes into the list of VMs using the session number. The even-numbered sessions will be re-routed to the VM with index 0 in the list, and the odd numbered sessions will be re-routed to the VM with index 1.

We use a clustered configuration for our IBM WebSphere and Apache Tomcat installations that enables sessions to be replicated across all live instances of the application servers. Session replication ensures that state information for each session is shared by all application servers in the cluster via peer-to-peer communication. This allows requests

[2]Recall that the filter must learn the various smoothing parameters used in the Holt-Winter model.

| Performance Metric | Risk Seeking ($\beta = -2$) | Risk Neutral ($\beta = 0$) | Risk Averse ($\beta = 2$) |
|---|---|---|---|
| Avg. Energy Savings | 37% | 41% | 41% |
| Avg. SLA Violations | 4,868 (0.01%) | 4,142 (0.01%) | 3,250 (0.01%) |
| Avg. Host Switching Activity | 76 | 66 | 60 |
| Avg. VM Migrations | 81 | 58 | 48 |
| Avg. Session Re-reroutes | 57 | 47 | 40 |

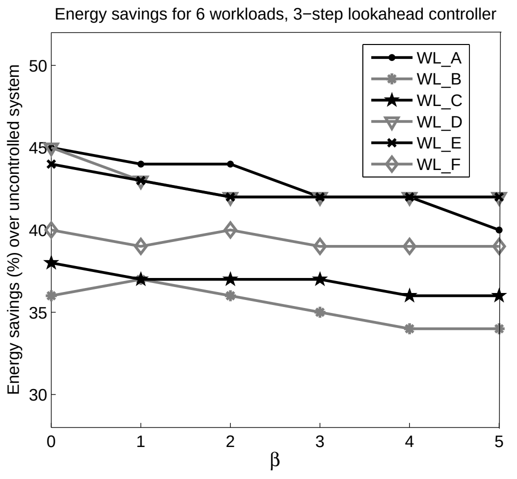
Figure 5.7: Comparison of controller performance operating in three regimes—risk-seeking ($\beta = -2$), risk-neutral ($\beta = 0$), and risk-averse ($\beta = 2$)—over a 24-hour period. Performance criteria include averages over 6 workloads for the energy savings over an uncontrolled system, the number of SLA violations, the host machine switching activity, the number of VM migrations, and the number of time sessions are re-routed due to a VM being shutdown.

belonging to a session to be re-routed between VMs with no interruption in service to the end user.

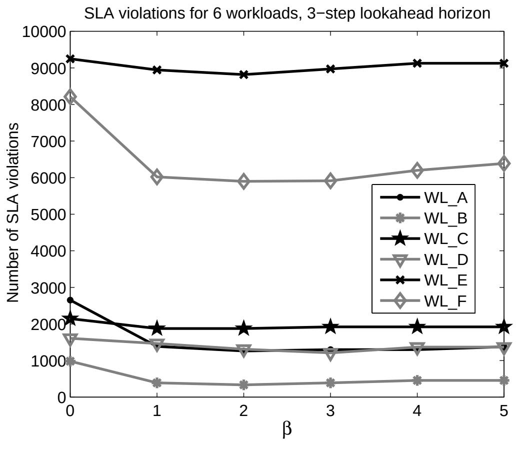
## 5.4 Performance of Risk-aware Control

This section examines the effect of tuning the risk-preference parameter $\beta$ on controller performance. Our objective is to identify a value for $\beta$ that maximizes energy savings and minimizes SLA violations. We first seek to verify the regime in which the controller performs best—risk seeking ($\beta < 0$), risk neutral ($\beta = 0$), or risk-averse ($\beta > 0$). Fig. 5.7 compares the controller performance within the three regimes, over six workloads, in terms of the energy savings achieved over an uncontrolled system during a 24-hour period, the number of SLA violations, the host machine switching activity[3], the number of live VM migrations, and the number of times sessions are re-routed from a de-allocated VM to backup VMs. Our experiments confirm what we have observed in previous work, that a risk-averse controller will outperform a risk-seeking controller in terms of energy savings and SLA violations, and with a reduction in switching activity [26]. The decrease in energy

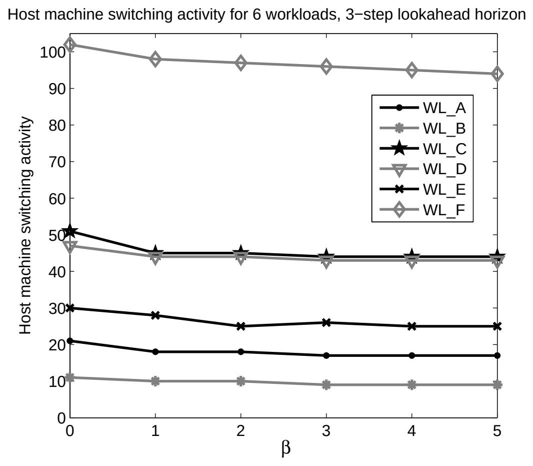
[3]Switching activity is defined as the number of times servers are powered up and down.



(a)



(b)



(c)

Figure 5.8: (a) The effect of the risk-preference value $\beta$ on the energy savings achieved for 6 workloads. As $\beta$ increases, the energy savings decrease slightly due to more conservative switching activity. (b) The effect of $\beta$ on SLA violations. As $\beta$ increases, the number of SLA violations decrease due to more conservative switching activity. (c) The effect of $\beta$ on switching activity. As $\beta$ increases, the switching activity correspondingly decreases.

savings with a risk-seeking controller is due to the controller's optimistic switching activity, which can reduce power savings under a noisy workload, incurring the extra costs of increased power consumption when powering up a host machine. Thus, we conclude that a risk-averse controller performs best under a noisy and time-varying enterprise workload.

Next, we tune the value of $\beta$ for a risk-averse control implementation. Figs. 5.8(a) shows the performance of risk-neutral ($\beta = 0$) and risk-averse ($\beta > 0$) controllers in terms of the energy savings achieved over an uncontrolled system during a 24-hour period. Although the energy savings decrease slightly, the number of SLA violations drops noticeably from $\beta = 0$ to $\beta = 2$, as shown in Fig. 5.8(b), an average reduction of about 30% when compared to the risk-neutral case. This is due to the conservative manner in which the risk-averse controller switches machines. Increasing $\beta$ also shows a steady decrease in server switching activity, as shown in Fig. 5.8. Around $\beta = 4$, however, the SLA violations begin to increase as the controller becomes overly risk-averse and begins to shun provisioning decisions. We conclude from Fig. 5.8 that to save energy and ensure a minimum number

| | Energy Savings 3-step Horizon | Energy Savings 4-step Horizon | Energy Savings 5-step Horizon |
|---|---|---|---|
| WL_A | 44% | 42% | 36% |
| WL_B | 36% | 32% | 34% |
| WL_C | 42% | 42% | 42% |
| WL_D | 42% | 44% | 40% |
| WL_E | 42% | 42% | 41% |
| WL_F | 40% | 39% | 34% |
| Avg. | 41% | 40% | 38% |

(a)

| | SLA Violations 3-step Horizon | SLA Violations 4-step Horizon | SLA Violations 5-step Horizon |
|---|---|---|---|
| WL_A | 1,263 | 3,330 | 4,024 |
| WL_B | 1,335 | 1,573 | 1,807 |
| WL_C | 1,874 | 1,425 | 1,947 |
| WL_D | 2,311 | 2,986 | 5,734 |
| WL_E | 8,818 | 8,765 | 10,056 |
| WL_F | 5,899 | 6,142 | 25,346 |
| Avg. | 3,583 | 4,036 | 8,152 |

(b)

Figure 5.9: (a) Energy savings of a risk-averse controller ($\beta = 2$) with increasing horizon length. Energy savings remain constant, on average, after increasing the prediction length from 3 to 4 steps, and decrease slightly at 5 steps. (b) SLA Violations of a a risk-averse controller ($\beta = 2$) with increasing horizon length. SLA violations increase when horizon length extends from $h = 3$ to $h = 4$, then increase again with $h = 5$ due to the increasing errors in the workload predictions.

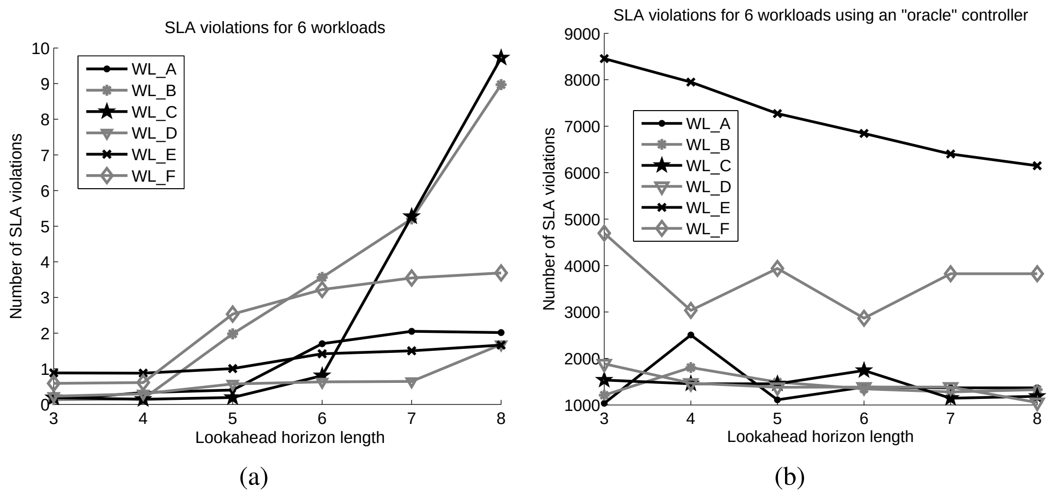


Figure 5.10: (a) The effect of horizon length on SLA violations for 6 workloads. The risk-preference in all cases is set to $\beta = 2$. The SLA violations show an increasing trend with horizon length due to the inaccuracies of the forecasting. (b) The effect of horizon length on SLA violations for 6 workloads using an "oracle" having perfect knowledge of the future. The risk-preference in all cases is set to a neutral $\beta = 0$ because there are no prediction errors. SLA violations generally show a decreasing trend with horizon length due to the accuracy of the forecasting.

of SLA violations, while reducing switching activity, a $\beta$ value of 2 is sufficient.

## 5.5 Effect of Tuning Horizon Length

Once we have established a best-performing value of $\beta$, we now study the effect of tuning the prediction-horizon length $h$ on the performance of a risk-averse controller with $\beta = 2$. The energy savings in Fig. 5.9(a) remain relatively constant when $h$ is 3 or 4, and decrease slightly when $h$ is 5. As seen from Fig. 5.9(b), the SLA violations increase slightly from $h = 3$ to $h = 4$, and again from $h = 4$ to $h = 5$. Therefore, we conclude that a prediction horizon of 3 steps is sufficient for good control performance in the general case. Also, longer horizon lengths can be explored at very little additional cost—the execution time of a 5-step lookahead controller remains at 2 seconds or less. However, as noted in Section 3.4, the prediction error typically increases with horizon length, and the probability of the controller making errant control decisions will increase with longer horizon lengths. Fig. 5.10(a) shows the increase in SLA violations as we extend the horizon further into the future to $h = 8$. By contrast, if the controller were to have perfect knowledge of future workload arrivals, the SLA violations would generally follow a decreasing trend as $h$ increases, as shown in Fig. 5.10(b).

## 5.6 Effect of User-Defined Policies

Excessive power cycling of a host machine can lead to reduced reliability in the form of software errors and disk failures [18, 58]. To control the repeated power cycling of servers, a system administrator may wish to specify a policy to leave a server powered on for a minimum time period once it is switched on, and such a policy acts as an additional operating constraint on the controller. The performance of a 3-step lookahead risk-averse controller with and without the policy is summarized in Fig. 5.11. We examine two policies, one to leave a server on for a minimum of 30 minutes once it is switched on, and the other to leave the server on for a minimum of 60 minutes. The impact of these policies is to reduce the number of SLA violations by 44% or more, by reducing the amount of switching activity by 58% or more, at the expense of a 2-4% decrease in power savings.

## 5.7 Time-varying Transaction Mix

Over a 24-hour run of the controller, we dynamically vary the transaction mix of the workload, about every 4 hours, to measure the effect on achieved response times and the number of SLA violations. The changing transaction mixes, comprised of browse/buy (or post) requests, are shown in Fig. 5.12(a). We use a 3-step lookahead, risk-averse controller ($\beta = 2$) with a policy to leave servers powered on for a minimum of 30 minutes. Using the system model captured in Figs. 4.4 through 4.6, we use linear interpolation to estimate the maximum processing rate achieved by a VM for any transaction mix, given some CPU

| | Energy Savings Risk-averse No Policy | Energy Savings Risk-averse 30-min. Policy | Energy Savings Risk-averse 60-min. Policy |
|---|---|---|---|
| WL_A | 42% | 43% | 43% |
| WL_B | 32% | 31% | 30% |
| WL_C | 42% | 42% | 41% |
| WL_D | 44% | 40% | 37% |
| WL_E | 42% | 42% | 38% |
| WL_F | 39% | 37% | 34% |
| Avg. | 41% | 39% | 37% |

(a)

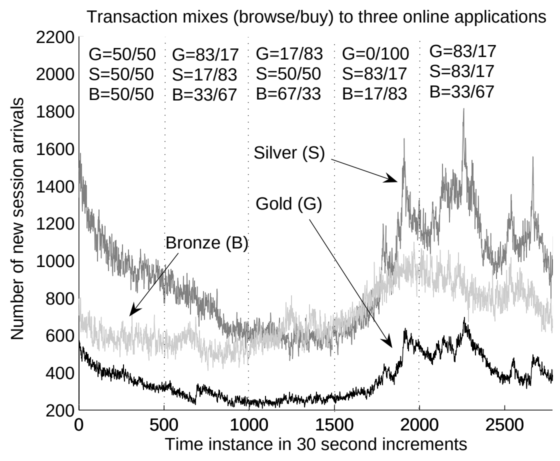
| | SLA Violations Risk-averse No Policy | SLA Violations Risk-averse 30-min. Policy | SLA Violations Risk-averse 60-min. Policy |
|---|---|---|---|
| WL_A | 1,263 | 1,199 | 654 |
| WL_B | 1,335 | 473 | 392 |
| WL_C | 1,874 | 1,425 | 137 |
| WL_D | 2,311 | 986 | 1,412 |
| WL_E | 8,818 | 2,251 | 907 |
| WL_F | 5,899 | 5,196 | 4,832 |
| Avg. | 3,583 | 2,021 | 1,389 |

(b)

| | Switching Activity Risk-averse No Policy | Switching Activity Risk-averse 30-min. Policy | Switching Activity Risk-averse 60-min. Policy |
|---|---|---|---|
| WL_A | 18 | 17 | 17 |
| WL_B | 10 | 9 | 9 |
| WL_C | 45 | 17 | 14 |
| WL_D | 44 | 15 | 13 |
| WL_E | 25 | 15 | 14 |
| WL_F | 97 | 26 | 22 |
| Avg. | 40 | 17 | 15 |

(c)

Figure 5.11: (a) Energy savings achieved by a risk-averse controller ($\beta = 2$) with policy implementation and a 3-step lookahead horizon. The average energy savings decrease slightly with the introduction of a policy to leave a server powered on for a minimum of 30 minutes. (b) The SLA violations decrease by an average of 44% or more with the introduction of a policy to leave a physical machine powered on for a minimum of 30 minutes. (c) The switching activity decreases by an average of 58% or more with the introduction of the same policies.



(a)

| Time period | Service | Avg. Response Time | Percent SLA Violations |
|---|---|---|---|
| $< 500$ | Gold | 141 | <0.01% |
| | Silver | 155 | <0.01% |
| | Bronze | 156 | <0.01% |
| 500-999 | Gold | 95 | <0.01% |
| | Silver | 197 | <0.01% |
| | Bronze | 130 | <0.01% |
| 1000-1499 | Gold | 176 | <0.01% |
| | Silver | 160 | <0.01% |
| | Bronze | 195 | <0.01% |
| 1500-1999 | Gold | 207 | <0.01% |
| | Silver | 121 | <0.01% |
| | Bronze | 124 | <0.01% |
| 2000-end | Gold | 102 | <0.01% |
| | Silver | 124 | <0.01% |
| | Bronze | 139 | <0.01% |

(b)

Figure 5.12: (a) Time-varying transaction mixes for the three services over a 24-hour period. (b) Average measured response times and measured SLA violations of a risk-averse ($\beta = 2$) controller with horizon length $h = 3$ and a policy to leave a host machine on for at least 30 minutes, where the transaction mix varies during each period as shown in Fig. 5.12(a). The results are summarized for each transaction mix within the 24-hour period.

share. If the transaction mix in Fig. 5.12(a) is dominated by browse requests, we interpolate between an all browse mix and a 50/50 mix. If the transaction mix is mostly buy (or post) requests, we interpolate between an all buy (post) mix and a 50/50 mix.

As Fig. 5.12(b) shows, the controller maintains system performance at an acceptable level even as the average response times vary with the transaction mix. As we would expect, the Bronze application (RUBBoS) in Fig. 4.6 shows higher response times for transaction mixes dominated by browse requests, due to the large database size. The Gold and Silver applications have higher response times for transaction mixes that are dominated by purchase requests.

## 5.8 Effects of Prediction Error

In an uncertain operating environment, controller decisions cannot be shown to be optimal since the controller does not have perfect knowledge of future environment inputs, and control decisions are made via a localized search within a limited prediction horizon. Therefore, we must be satisfied with good sub-optimal decisions.

Our final series of tests were aimed at comparing a practical controller implementation against an "oracle" that has perfect knowledge of future workload arrivals. We noted that even if workload predictions are completely error free, the energy savings are about the same, and the number of SLA violations are lower than in the practical case where only imperfect predictions can be obtained. We selected a best-performing risk-averse ($\beta = 2$) controller having a 3-step lookahead horizon and compared it against a 3-step oracle controller. Figs. 5.13(a) and 5.13(b) show that having perfect workload predictions does not greatly affect energy savings and reduces SLA violations by an average of 27%. The improved performance of the oracle controller should be expected, considering that the average error in the Kalman estimates starts at 5% and increases 1% for each prediction step thereafter.

| | Energy Savings Kalman Predictions | Energy Savings "Oracle" Predictions |
|---|---|---|
| WL_A | 42% | 44% |
| WL_B | 32% | 38% |
| WL_C | 42% | 36% |
| WL_D | 44% | 37% |
| WL_E | 42% | 42% |
| WL_F | 39% | 35% |
| Avg. | 41% | 39% |

(a)

| | SLA Violations Kalman Predictions | SLA Violations "Oracle" Predictions | Percent Reduction |
|---|---|---|---|
| WL_A | 1,263 | 1,105 | 13% |
| WL_B | 1,335 | 1,342 | -1% |
| WL_C | 1,874 | 1,456 | 22% |
| WL_D | 2,311 | 1,469 | 36% |
| WL_E | 8,818 | 7,270 | 18% |
| WL_F | 5,899 | 3,034 | 47% |
| Avg. | 3,583 | 2,613 | 27% |

(b)

Figure 5.13: (a) Energy savings for a 3-step lookahead, risk-averse ($\beta = 2$) controller using workload predictions with and without errors. The average energy saving is about the same for a controller with and without perfect workload predictions. (b) SLA violations. Perfect predictions from the Kalman filter reduces SLA violations by an average of 27%.

## 5.9 Summary

The experiments detailed in this chapter show that over a 24-hour period, the implemented LLC framework conserves, on average, 41% of the energy consumed by an uncontrolled system. A risk-averse controller with $\beta = 2$ reduces SLA violations as well as the switching activity, with a marginal loss in energy savings as compared to a controller with no risk preference ($\beta = 0$). A slightly risk-averse 3-step controller, with a policy to leave servers powered on for a minimum of 30 or 60 minutes performs the best over all the controller configurations that were tested, in terms of energy savings, SLA violations, and switching activity.

# 6. Controller Scalability

This chapter develops some techniques aimed at improving the scalability of the controller to extend the framework to larger data centers. In particular, we discuss how approximation theory can be applied to decrease the computational complexity of provisioning a large-scale system. We present some preliminary data on the execution time of the control framework using a neural network approximating structure as we expand the size of the system. Finally, we explain how the centralized control architecture can be converted to a distributed hierarchy.

## 6.1 Introduction

Scalability is of concern for the control framework presented in this thesis to be of practical use in large-scale data centers. The discrete state-space nature of limited lookahead control, although well-suited to characteristics of the control problem of interest, does not easily scale to handle large-scale computing systems. When control inputs must be chosen from discrete values, the LLC problem in (4.18) will show an exponential increase in worst-case complexity with an increasing number of control options and longer prediction horizons—the so-called "curse" of dimensionality. The control space for our system of interest grows exponentially in $n$, the number of VMs to provision, and $N$, the number of host machines to operate. Thus, the size of the search space poses a challenge for centralized controller implementations to provide fast resource-provisioning decisions.

There are several methods to improve the practical value of the online optimization scheme developed in this thesis for large-scale settings.

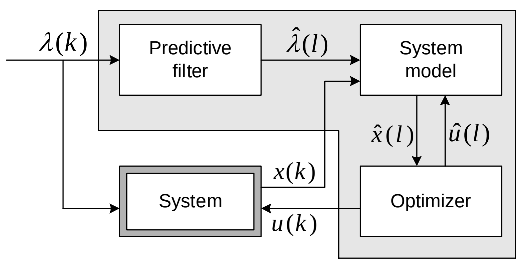
- *Linear trajectory exploration.* As discussed in Section 4.4, the 'holding-pattern' introduced to the cost estimation effectively reduces exponential complexity to linear complexity in horizon length, $h$.

- *Multi-threaded implementations of the controller.* Multiple threads can parallelize the optimization problem over distributed processing nodes or across CPU cores in a multi-core machine to expedite the controller execution time.
- *Applied approximation theory.* Concepts from approximation theory can be applied within the control framework to further reduce the computational burden of controlling large-scale systems. The relevant approximations are made in the optimization of the control variables input to the system.
- *Distributed hierarchical control.* The overall problem can be decomposed into a set of simpler sub-problems and solved in cooperative fashion by multiple controllers.

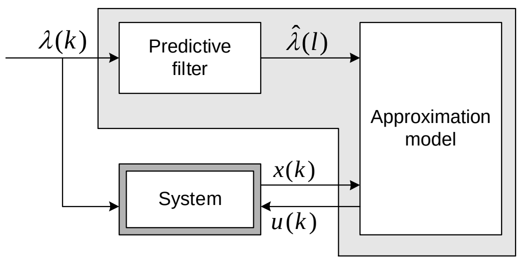
We present preliminary data of applying approximation theory in the form of a neural network to the control framework to reduce the execution time of the controller by 80%, with only a 3% sacrifice in controller performance. We then discuss how the centralized framework may be made modular to develop larger, distributed hierarchical control structures. Our preliminary work on the use of approximating structures in distributed computing environments has been published in [28] and [29].

## 6.2 Control Action Approximation

As system grows in scale, it is advantageous to train an approximation model to predict the behavior of the controller. By learning the control action under a variety of conditions, two components in the LLC framework in Fig. 6.1(a) can be replaced with one approximation model as shown in Fig. 6.1(b), effectively eliminating the combinatorial search process within the optimizer. By learning the control action, the time-intensive combinatorial search process can be eliminated from the control process to make the infrastructure more scalable. In preliminary scalability studies, we use a neural network (NN) to learn



(a)



(b)

Figure 6.1: (a) The schematic of a limited lookahead controller. (b) A modified control schematic of Fig. (a) showing replacement of the system model and optimizer with an approximation model.

the tendencies of the controller in Fig. 6.1(a), in terms of its decision making, via offline simulations. At run time, given the current state and environment inputs, a NN placed into Fig. 6.1(b) provides an approximate solution, which is used as a starting point around which to perform a local search of $\pm$ 1 VM per host to obtain the final control decision.

Approximation structures such as neural networks and regression trees can help reduce the difficulties of controlling a complex non-linear computing system and reduce the associated control overhead. The key to developing an approximation structure is to fit measurements obtained from the controlled system, i.e., training data, to an abstract mathematical representation that accepts one or more features from each training point as input and provides an estimate of system behavior (e.g., response time) as the output. Two types of approximating structures—neural networks and regression trees—are described as follows.

**Neural Networks**. Applications in economics, engineering, pure science and gaming often apply neural networks to represent nonlinear systems for estimation and forecasting purposes. Neural networks are abstract mathematical models that are trained offline using simulation data to predict one or more outcomes from an input vector. Neural networks are a good choice for behavioral approximation when the exact relationship between inputs and outputs of a system are impossible or difficult to define, such as in complex, nonlinear, delayed or high-dimensional systems. Although typically applied to data in the continuous domain, neural networks can accommodate the discrete domain with some quantized output filtering.

The basic architecture of a neural network consists of an input layer, an output layer and one or more hidden layers. The hidden layers introduce additional nonlinearity between the input and output layers and are added as needed. Input values $\{x_1, \ldots x_Z\}$ are multiplied by some scalar weight, and to that product an input bias may be added to raise positive values and decrease negative values. The weighted sum of the input layers is then passed to an activation function which may be a linear, sigmoid, or other nonlinearly shaped function. The activation function limits the amplitude of an output. Adjustable parameters of a neural network include the initial input weights, the presence of input biases, the number of hidden layers.

The general form of a fully-connected neuron consists of units of one layer connected to all units of the next layer. The feedforward neural network passes the output $y$ of one layer as the input to the next layer [16]. Backpropagation within a neuron enables updating of the synaptic (edge) weights $w$ and biases $b$ during network training. Mathematically, the output of a neuron can be expressed as:

$$y = \psi\big(\sum_{z=1}^{Z} x_z \cdot w_z + b\big) \tag{6.1}$$

During the training phase, updates upon the edge weights seek to minimize the squared error between the neural network's estimated output and the desired output. The neural network becomes increasingly nonlinear as the edge weights are updated. Neural networks typically make use of an additional parameter, a momentum factor, to prevent getting stuck at local minima and terminating the training process prematurely. The advantage of neural networks over other models is that they can be updated online in response to changing dynamics in the system.

**Regression Trees**. Regression trees, sometimes referred to as classification trees, are a form of recursive partitioning that can be represented as a tree. Recursive partitioning is particularly well-suited to discrete data and nonlinear system dynamics [14]. The tree construction process sorts historic data points so as to maximize statistical criteria for one or more predicated values of a chosen system variable at a time. The partitioning function is called recursively, then ceases when there are fewer than $K$ training points to be partitioned. Tree pruning collapses two or more regions into one for a more compact representation. A regression tree representation is typically less compact than that of a neural network. Regression trees are static models in the sense that they are generated offline and must be reconstructed when changes in the system behavior or structure occur. Mathematically, the output of a regression tree can be expressed as [57]:

$$y = \sum_{d=1}^{D} c_d I\big(x \in Region_d\big) \tag{6.2}$$

$$\textbf{where:} \quad x = \big\{x_1,\, \dots x_z\big\}$$

where $D$ denotes the total number of mappable regions or possible output values. Recursive partitioning assigns the values of $Z$ input variables $x = \big\{x_1,\, \dots x_Z\big\}$ into $D$ regions $\big(Region_1, \dots Region_D\big)$, scaling the incidence $I \in \{0, 1\}$ that an input vector falls within a particular region by the output value $c_d$. Input values of selected system variables dictate a path through the tree until terminating at a leaf node, resulting in a predicted output.

Initial results using a back-propagation, feed-forward NN, implemented using the Matlab Neural Net Toolbox, reduce the execution time of a 4-step lookahead controller for 10 host machines to about 30 seconds, as shown in Fig. 6.2[1]. The network is trained using a set of 1,000 training points collected over two workloads similar to that shown in Fig. 3.6.

[1]Neural net results are pending for a system size of 15 hosts.

| Controllable System Size | Control Options | Avg. LLC Exe. Time | Avg. NN Exe. Time |
|---|---|---|---|
| 5 Hosts | 1x$10^3$ | $<$ 2 sec. | – |
| 10 Hosts | 1x$10^6$ | 150 sec. | 30 sec. |
| 15 Hosts | 1x$10^9$ | 30 min. | – |

Figure 6.2: Execution times incurred by the controller for larger clusters at the application tier.

While the energy savings of the neural network controller for a sample workload decrease slightly, from 29% to 27%, as compared to the exploratory LLC controller, the number of SLA violations improves, from 5,116 to 4,502. Further refinements of the neural network and a larger training data set will help to improve the performance of the controller.

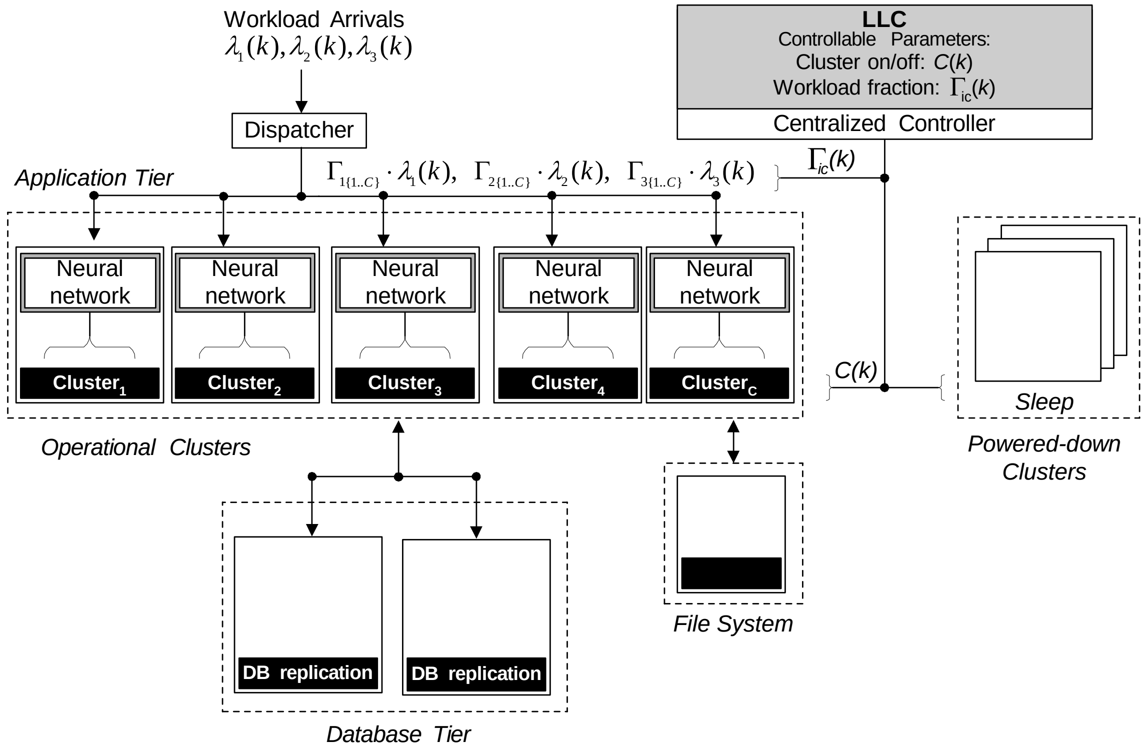


Figure 6.3: The system architecture supporting the Gold, Silver, and Bronze services. The controller sets $C(k)$, the number of active clusters, and $\Gamma_{ic}(k)$, the fraction of workload to distribute to the $c^{th}$ cluster. Powered-off clusters are denoted in Sleep.

Once the neural network has been properly trained and its performance verified, we can then view the entire cluster as a self-managing container. Provided that the larger system can be partitioned into such clusters of identical size and processing capacity, we then propose to construct a distributed, hierarchical controller, as shown in Fig. 6.3. Each cluster is managed by a neural network, and at the top level, an exploratory LLC controller communicates with the lower-level controllers, viewing a cluster as the smallest computing unit within the system. The top-level controller then decides $C(k)$, the number of clusters to operate at time $k$, and $\Gamma_{ic}(k)$, the fraction of the $i^{th}$ workload to distribute to the $c^{th}$ cluster. The cluster container is put in a Sleep state when the computing resources are not needed.

# 7. Extensions to the Current Work

Looking ahead to extensions of this thesis, we are primarily concerned with the controller's reliance upon static models of application behavior that do not adapt to slow changes that can occur in production, and those introduced by software errors. By introducing adaptive models that dynamically adjust to the changing performance profile of an application, a robust controller can maintain performance objectives over time. In this chapter, we motivate the need for adaptive models, characterizing the effects of some events that change an application's behavior.

## 7.1 Introduction

Our experience with the experimental cluster in Fig. 3.3 suggests that long-running software components managed by a controller be treated as *adaptive processes* that exhibit slow behavioral changes over time as a result of both internal and external influences [44]. For example, consider the Trade6 application. The response time achieved by a long-running Trade6 application can be time varying even under a constant workload intensity due to: (1) internal changes to the application as it "ages" (e.g., slow performance degradation caused by memory leaks) and/or (2) external actions wherein users and stocks are added to the database or removed from it. This time-varying behavior has significant implications on model-based control techniques such as LLC, in that, if a static model, such as that shown in Fig. 4.5 is used to predict the response time achieved by Trade6, then, over time, the predictions will not match the actual response times. In this chapter, we motivate the need for model-adaptive control to maintain system power and performance objectives over time under dynamic operating conditions.

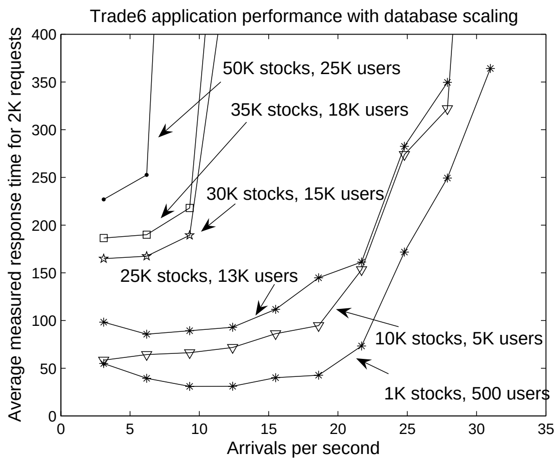
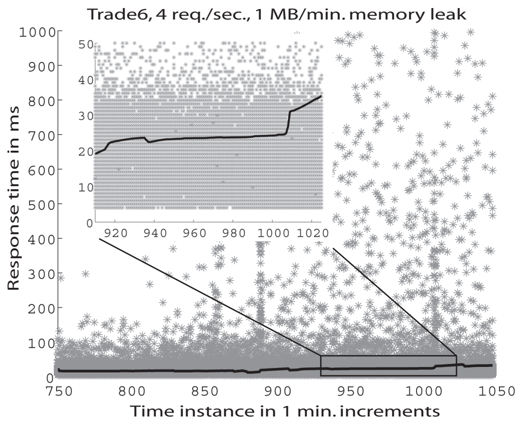


Figure 7.1: The changing performance profile of the Trade6 application as the database size grows.

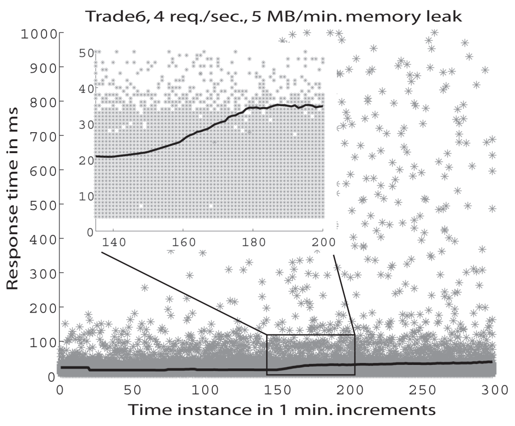
## 7.2 Software Components as Adaptive Processes

To study the problem of software behaving as an adaptive process, we first sought to characterize the effects of database file size growth on an application performance. The size of an application's database may increase due to user activity over the lifetime of the online service as new transactions accumulate. The database size affects the retrieval time for dynamic content, as well as the time to commit new transactions. Fig. 7.1 clearly shows the effects of database size on the average response time per request for the Trade6 application. The default database for the Trade6 application consists of 500 users and 1,000 stocks and is scaled upward from that size. As the size of the database grows, the maximum workload arrival rate that can be tolerated within the bounds of the SLA shown in Fig. 3.2 decreases. An adaptive online controller would scale back the workload directed to each node to accommodate the increased response time per transaction.

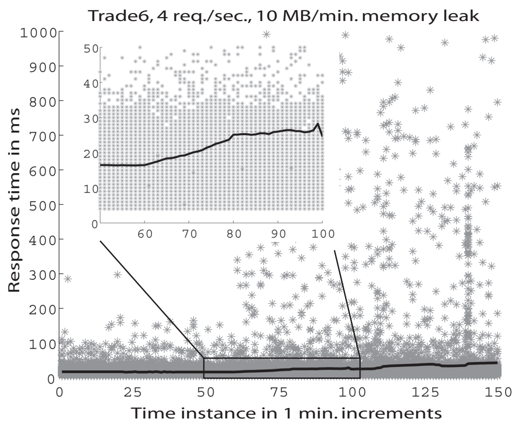
While growing database sizes can be considered an effect of normal operation, memory leaks caused by software bugs are a common source of errant operation in an application, particularly when using the C/C++ programming languages [49]. A memory leak is defined



(a)



(b)

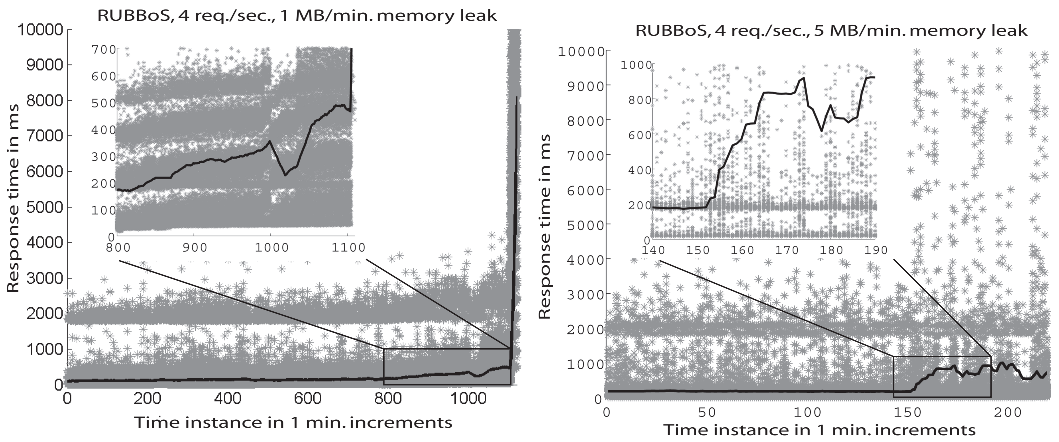


(c)

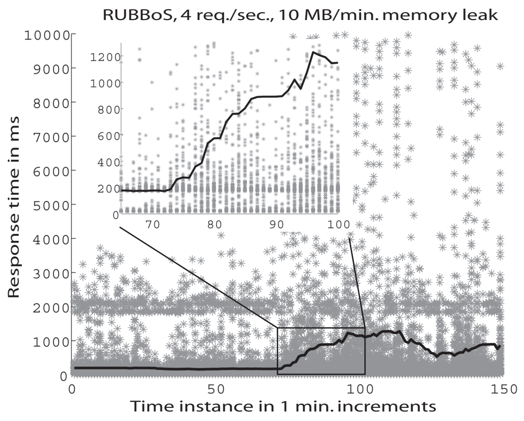
Figure 7.2: The measured response times for the Trade6 application as (a) 1 MB , (b) 5 MB, and (c) 10 MB memory leaks accumulate per minute. The VM is allocated a 6 GHz CPU share and 1 GB of memory.

as memory that has been dynamically allocated by an application, such as by using the $malloc()$ operation in the C language, but that is never released back to the operating system. Memory leaks that accumulate will eventually allocate the resource to capacity and the operating system will reserve the swap space (virtual memory). It is typically at this point when the application has reserved the system memory to capacity that performance significantly degrades.

Figs. 7.2 and 7.3 show the response times measured per-transaction for the Trade6 and



(a) (b)



(c)

Figure 7.3: The measured response times for the RUBBoS application as (a) 1 MB, (b) 5 MB, and (c) 10 MB memory leaks accumulate per minute. The solid line indicates the average response time per one minute increment. The VM is allocated a 6 GHz CPU share and 1 GB of memory.

RUBBoS applications, respectively. The solid black lines through Fig. 7.2 and Fig. 7.3 indicate the response time averaged over the last 20 time samples. In both applications, the memory leak was effected at the application server and database tiers and the workload arrival rate was held constant. The point at which application performance degrades significantly is commensurate with the rate of the memory leak, occurring at about $t = 150$, or two and a half hours into a 5 MB per minute memory leak, and at half that time, occurring at about $t = 70$, or an hour and fifteen minutes into a 10 MB per minute memory leak. The 1 MB per minute memory leak shown in Fig. 7.3(a) exhibits a more gradual increase before causing the application server to crash, at about $t = 1100$.

The number of per-transaction SLA violations, those requests that exceed the acceptable response time thresholds, jumps from less than one hundredth of a percent to $2 - 5\%$ of the total requests over a given time period. Because SLA goals are set somewhat conservatively, the percentage of per-transaction SLA violations may not be large. However, the increases in average response times are a more dramatic measure of performance degradation. The Trade6 application exhibits somewhat modest increases in average response times (about 60% to 75% above normal) as the memory is allocated to capacity.

## 7.3 The Fault-adaptive Control Framework

To make good provisioning decisions, an online controller must maintain an accurate model of application behavior under various workload intensities when hosted by a VM given a particular CPU and memory allocation. In previous work, we have captured this behavior using supervised learning, i.e., by extensively simulating the VM (and the underlying application) in offline fashion and using the observations to train an approximation structure such as a neural network for use at run time [28]. However, it is generally not possible, via offline simulations, to create an exact model of system dynamics using a limited set of training data. Therefore, we propose future work to integrate online parameter tuning and model-learning techniques within the LLC control framework to: (1) improve the accuracy of partially specified system models, and (2) maintain the correctness of the model against slow behavioral changes to system components over time.

We can use supervised online learning to construct (approximate) dynamical models of the application components under control. These models capture the complex and nonlinear relationship between the response time achieved by a VM, and the CPU and memory

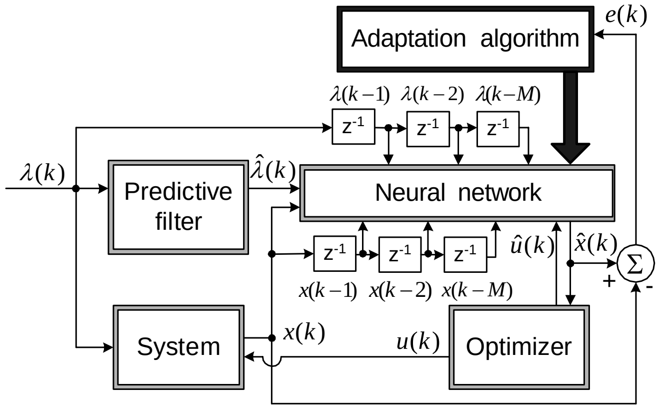


Figure 7.4: The proposed fault-adaptive control framework utilizing a recurrent neural network

share provided to it, under dynamically changing operating conditions. We propose to incorporate the control approach developed in this thesis with an online model-learning component to create a fault-adaptive control framework as shown in Fig. 7.4, aimed at developing autonomic systems that anticipate changes in operating conditions and react to gradual behavior changes in the system components.

Fig. 7.4 shows the adaptive components integrated with the LLC framework, using a recurrent neural network for the System Model [1]. Re-training of the neural network utilizes datasets of length $M$, where the previous environment estimates and system state estimates are retained from time $(l-1)$ through time $(l-M)$, using blocks denoted as $Z^{-1}$ for the time delay. A comparator, denoted as $\sum$, inputs the actual state of the system, $x$ and compares it against the estimated state of the system as output by the neural network, denoted as $\hat{x}$. The error, $e(x)$, is then applied to a model adaptation algorithm. The process by which the model is updated will be dependent upon the modeling approach chosen. For example, for a simple lookup table, the adaptation process is a matter of updating the table entries. For abstract models, such as the recurrent neural network in Fig. 7.4, the adaptation process involves re-training the neural network as new data becomes available. The cost in terms of the time to perform the adaptation will depend upon the length $M$ of the datasets.

The key challenge is to learn the new system behavior under dynamic conditions and update the model used by the controller in real time. An accurate system model helps to avoid the excessive switching a controller will introduce into the system to correct for misguided control actions, and to reduce the number of SLA violations. The key challenge will be for the model adaptation unit to learn an accurate new model of the system in a timely manner. As shown in Fig. 7.3, changes in the application behavior can occur suddenly. To tackle this problem, it will be useful to study several modeling techniques (non-linear regression, auto-regressive moving averages, etc.) for their estimation accuracy and convergence time.

# 8. Conclusion

We have presented an optimization framework for power and performance management in a virtualized computing system hosting multiple online services with session-based workloads. The LLC scheme aims to achieve higher server utilization and energy efficiency by dynamically provisioning VMs, consolidating the workload, and turning servers on and off as needed. It also accommodates the switching costs of provisioning decisions, and explicitly encodes the risk associated with making these decisions in an uncertain operating environment. We also show how user-defined policies can serve as additional operating constraints to the controller, specifically to reduce the power cycling of servers.

We experimentally validate the LLC scheme on a server cluster supporting three online services, and show that the cluster, when managed using a well-tuned controller, saves, on average, 41% in power-consumption costs over a twenty-four hour period when compared to a system without dynamic control. The control problem accounts for the switching costs of provisioning decisions, and explicitly encodes the risk associated with making these decisions in an uncertain operating environment. Our results indicate that a risk-averse controller can reduce SLA violations by about 30% as compared to a controller without any risk preference. We also show how user-defined policies can serve as additional operating constraints to the controller, specifically to reduce the power cycling of servers. The impact of these policies is to reduce the number of SLA violations, by reducing the amount of switching activity by 58% or more, at the expense of a 2-3% decrease in power savings.

We also use concepts from approximation theory to further reduce the computational burden of controlling larger systems, and show that a neural-network based scheme can reduce the execution time of a non-approximating controller by 80%, with only a 3% sacrifice in energy savings. Finally, it is hoped that the framework developed in this thesis will advance the conversion from manual and ad-hoc control approaches to automated control processes based on sound theoretical concepts and concrete mathematical models. The model-based techniques presented in this thesis will simplify the development of a large class of distributed computing systems, enabling those systems to operate in a performance optimized, energy efficient manner.

# Bibliography

[1]

[2] ABDELWAHED, S., KANDASAMY, N., AND NEEMA, S. Online control for self-management in computing systems. In *Proc. of Real-Time and Embedded Technology and Applications Symposium (RTAS)* (May 2004), IEEE, pp. 368–375.

[3] ABDELZAHER, T. F., SHIN, K. G., AND BHATTI, N. Performance guarantees for web server end–systems: A control theoretic approach. *IEEE Trans. Parallel & Distributed Syst. 13*, 1 (January 2002), 80–96.

[4] ARLITT, M., AND JIN, T. Workload characterization of the 1998 world cup web site. Tech. rep., Hewlett-Packard Labs, Technical Report HPL-99-35R1, Sept. 1999.

[5] BOARDMAN, B., DOHERTY, S., JR., C. F., MACVITTIE, D., AND WILSON, T. This old data center. *Network Computing* (May 26 2005), 28.

[6] CERVIN, A., EKER, J., BERNHARDSSON, B., AND ARZEN, K. Feedback-feedforward scheduling of control tasks. *J. Real-Time Syst. 23*, 1-2 (2002), 25–53.

[7] COPELAND, T., AND WESTON, J. *Financial Theory and Corporate Policy, 3rd, ed.* Addison-Wesley, 1988.

[8] DAREMA, F. Grid computing and beyond: The context of dynamic data driven applications systems. *Proc. of the IEEE 93*, 3 (March 2005), 692–97.

[9] DELL. *Dell DVD Store Database Test Suite*. http://linux.dell.com/dvdstore.

[10] GANEK, A. Overview of autonomic computing: Origins, evolution, direction. In *Autonomic Computing: Concepts, Infrastructure and Applications* (2007), CRC Press, pp. 3–18.

[11] GANEK, A., AND CORBI, T. The dawn of the autonomic computing era. *IBM Systems Journal 41*, 1 (Sept. 2003), 5–18.

[12] GOŠEVA-POPSTOJANOVA, K., LI, F., WANG, X., AND SANGLE, A. A contribution towards solving the web workload puzzle. In *IEEE Conf. on Dependable Systems and Networks* (Jun. 2006), pp. 505–514.

[13] HADJ-ALOUANE, N. B., LAFORTUNE, S., AND LIN, F. Variable lookahead supervisory control with state information. *IEEE Trans. Autom. Control 39*, 12 (Dec. 1994), 2398–2410.

[14] HARELL, F. *Regression Modeling Strategies*. Springer, 2001.

[15] HARVEY, A. *Forecasting, Structural Time Series and the Kalman Filter*. Cambridge University Press, 1989.

[16] HAYKIN, S. *Neural Networks: A Comprehensive Foundation*. Prentice-Hall, 1999.

[17] HELLERSTEIN, J. L., DIAO, Y., PAREKH, S., AND TILBURY, D. M. *Feedback Control of Computing Systems*. Wiley-IEEE Press, 2004.

[18] HUGHES, G., AND MURRAY, J. Reliability and security of raid storage systems and d2d archives using sata disk drives. *ACM Trans. on Storage 1* (Feb. 2005), 95–107.

[19] IBM. An architectural blueprint for autonomic computing. Tech. rep., IBM Research Report, Thomas J. Watson Research Center, Oct. 2004.

[20] IBM. *Trade6 Performance-Characterizing Application for WebSphere*. http://www.ibm.com/developerworks/edu/dm-dw-dm-0506lau.html, 2005.

[21] INTEL CORP. *Enhanced Intel SpeedStep Tecnology for the Intel Pentium M Processor*, 2004.

[22] KANDASAMY, N., ABDELWAHED, S., AND HAYES, J. Self-optimization in computer systems via on-line control: Application to power management. In *Proc. IEEE Intl. Conf. on Autonomic Computing (ICAC)* (May 2004), IEEE, pp. 54–61.

[23] KEPHART, J., CHAN, H., LEVINE, D., TESAURO, G., RAWSON, F., AND LEFURGY, C. Coordinating multiple autonomic managers to achieve specified power-performance tradeoffs. In *Proc. IEEE Intl. Conf. on Autonomic Computing (ICAC)* (Jun. 2007), pp. 145–154.

[24] KEPHART, J., AND CHESS, D. The vision of autonomic computing. *IEEE Computer 36* (Jan. 2003), 41–50.

[25] KHANNA, G., BEATY, K., KAR, G., AND KOCHUT, A. Application performance management in virtualized server environments. In *Proc. of the IEEE Network Ops. and Mgmt. Sym.* (Apr. 2006), pp. 373–381.

[26] KUSIC, D., AND KANDASAMY, N. Risk-aware limited lookahead control for dynamic resource provisioning in enterprise computing systems. In *Proc. IEEE Intl. Conf. on Autonomic Computing (ICAC)* (June 2006), pp. 74–83.

[27] KUSIC, D., AND KANDASAMY, N. Risk-aware limited lookahead control for dynamic resource provisioning in enterprise computing systems. *J. of Cluster Computing 10* (Aug. 2007), 395–408.

[28] KUSIC, D., KANDASAMY, N., AND JIANG, G. Approximation modeling for the online performance management of distributed computing systems. In *Proc. IEEE Intl. Conf. on Autonomic Computing (ICAC)* (June 2007), p. 23.

[29] KUSIC, D., KANDASAMY, N., AND JIANG, G. Approximation modeling for the online performance management of distributed computing systems. *IEEE J. of Systems, Man and Cybernetics-B 38* (Oct. 2008), 1221–1233.

[30] KUSIC, D., KEPHART, J., HANSON, J., KANDASAMY, N., AND JIANG, G. Power and performance management of virtualized computing environments via lookahead control. In *Proc. IEEE Intl. Conf. on Autonomic Computing (ICAC)* (Jun. 2008), pp. 3–12.

[31] LEFURGY, C., WANG, X., AND WARE, M. Server-level power control. In *Proc. IEEE Conf. on Autonomic Computing* (Jun. 2007), p. 4.

[32] LI, F., GOŠEVA-POPSTOJANOVA, K., AND ROSS, A. Discovering web workload characteristics through cluster analysis. In *IEEE Sym. on Network Computing and Applications* (Jul. 2007), pp. 61–68.

[33] LIU, X., ZHU, X., SINGHAL, S., AND ARLITT, M. Adaptive entitlement control of resource containers on shared servers. In *9th IFIP/IEEE Int'l Symp. Integrated Network Management (IM)* (2005), pp. 163–176.

[34] LU, C., ALVAREZ, G. A., AND WILKES, J. Aqueduct: Online data migration with performance guarantees. In *Proc. USENIX Conf. File Storage Tech.* (2002), pp. 219–230.

[35] LU, C., STANKOVIC, J., TAO, G., AND SON, S. Feedback control real-time scheduling: Framework, modeling and algorithms. *Journal of Real-Time Systems 23*, 1-2 (July/September 2002), 85–126.

[36] LU, Z., HEIN, J., HUMPHREY, M., STAN, M., LACH, J., AND SKADRON, K. Control-theoretic dynamic frequency and voltage scaling for multimedia workloads. In *Proc. Int'l Conf. Compilers, Architectures, & Synthesis Embedded Syst. (CASES)* (2002), pp. 156–163.

[37] MACIEJOWSKI, J. M. *Predictive Control with Constraints*. Prentice Hall, London, 2002.

[38] MASCOLO, S. Classical control theory for congestion avoidance in high-speed internet. In *Proc. Conf. Decision & Control* (1999), pp. 2709–2714.

[39] MCKINSEY. *Revolutionizing Data Center Efficiency*. McKinsey & Company, http://uptimeinstitute.org, 2008.

[40] MENASCÉ, D., AND ALMEIDA, V. A. F. *Capacity Planning for Web Services: Metrics, Models, and Methods*. Prentice Hall PTR, Upper Saddle River, NJ, 2002.

[41] MENASCÉ, D., ET AL. In search of invariants for e-business workloads. In *Proc. ACM Conf. Electronic Commerce* (2000), pp. 56–65.

[42] MICHALOPOULOS, D. New applications: Device cuts computer energy consumption. *Wired Magazine 9* (Aug. 1976).

[43] MURCH, R. *Autonomic Computing*. Prentice-Hall, 2004.

[44] OREIZY, P., MEDVIDOVIC, N., AND TAYLOR, R. Runtime software adaptation: framework, approaches, and styles. In *ACM Int'l Conf. on Software Engineering* (May 2008), pp. 899–910.

[45] PAREKH, S., GANDHI, N., HELLERSTEIN, J., TILBURY, D., JAYRAM, T., AND J.BIGUS. Using control theory to achieve service level objectives in performance management. In *Proc. IFIP/IEEE Int. Symon Integrated Network Management* (May 2001), pp. 841–54.

[46] PINHEIRO, E., BIANCHINI, R., AND HEATH, T. *Dynamic Cluster Reconfiguration for Power and Performance*. Kluwer Academic Publishers, 2003.

[47] RANGANATHAN, P., LEECH, P., IRWIN, D., AND CHASE, J. Ensemble-level power management for dense blade servers. In *Proc. of the IEEE Sym. on Computer Architecture* (Jun. 2006), pp. 66–77.

[48] REISS, S. Cloud computing, available at amazon.com today. *Wired Magazine 16* (Apr. 2008).

[49] RICHARDS, J. Software trends for the 21st century. *Computer Weekly* (Jun. 6, 2006).

[50] RICKWOOD, L. Google apps: Killer software or killer decision? *PCworld.ca* (Mar. 2007).

[51] ROLIA, J., CHERKASOVA, L., ARLITT, M., AND ANDRZEJAK, A. A capacity management service for resource pools. In *Proc. of the ACM Wkshp. on Software and Perf.* (Jul. 2005), pp. 229–237.

[52] RUBBOS. *RUBBoS: Bulletin Board Benchmark*. http://jmob.objectweb.org/rubbos.html.

[53] SIMUNIC, T., AND BOYD, S. Managing power consumption in networks on chips. In *Proc. Design, Automation, & Test Europe (DATE)* (2002), pp. 110–116.

[54] SMITH, R. Power companies consider options for energy sources. *The Wall Street J.* (Oct. 29 2007), A.10.

[55] STEINDER, M., WHALLEY, I., CARRERA, D., GAWEDA, I., AND CHESS, D. Server virtualization in autonomic management of heterogeneous workloads. In *Proc. of the IEEE Sym. on Integrated Network Management* (May 2007), pp. 139–148.

[56] TSAI, C., SHIN, K., REUMANN, J., AND SINGHAL, S. Online web cluster capacity estimation and its application to energy conservation. *IEEE Trans. on Parallel and Dist. Sys. 18*, 7 (Jul. 2007), 932–945.

[57] WASSERMAN, L. *All of Nonparametric Statistics*. Springer, 2006.

[58] WEDDLE, C., OLDHAM, M., QIAN, J., WANG, A., REIHER, P., AND KUENNING, G. Paraid: A gear-shifting power-aware raid. *ACM Trans. on Storage 3* (Nov. 2007), 13.

[59] WELSH, M., AND CULLER, D. Adaptive overload control for busy internet servers. In *Proc. of USENIX Sym. on Internet Technologies and Systems (USITS)* (March 2003).

[60] WHITE, R., AND ABELS, T. Energy resource management in the virtual data center. In *Proc. IEEE Intl. Conf. on Autonomic Computing (ICAC)* (May 2004), pp. 112–116.

[61] XU, J., ZHAO, M., FORTES, J., CARPENTER, R., AND YOUSIF, M. On the use of fuzzy modeling in virtualized data center management. In *Proc. IEEE Intl. Conf. on Autonomic Computing (ICAC)* (Jun. 2007), pp. 25–35.

# Vita

Dara Kusic was born in Pittsburgh, Pennsylvania, U.S.A. She received a B.S. (Summa Cum Laude) in Computer Engineering from the University of Pittsburgh in 2003, and an M.S. in Electrical Engineering from the University of Pittsburgh in 2005. Dara also holds a B.A. in Urban Studies from the University of Pennsylvania. Her Ph.D. work at Drexel University has focused on developing self-managing computing systems that optimize their operation for power and performance management goals. She spent the summer of 2007 working as an intern at the IBM T.J. Watson Research Laboratories in Hawthorne, New York.

She is a recipient of the ECE Department Graduate Student Fellowship, the AAAS Mass Media Fellowship, the NSF GK-12 Fellowship, as well as two best paper awards at the IEEE International Conference on Autonomic Computing in 2006 and 2008. She is a member of the IEEE and the IEEE Computer Society.