PERFORMANCE COMPOSABILITY: AN EMERGING CHALLENGE IN PERFORMANCE-ADAPTIVE SYSTEMS

BY

JIN HEO

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Abstract

The goal of this dissertation is to design, implement, and evaluate design techniques and software support for performance composition and dynamic performance management in large-scale performance-sensitive systems. The topic is motivated by the complexity of large-scale software systems due to increased use of automated adaptation policies in lieu of manual performance tuning.

Composition of multiple adaptive components in large-scale software systems, however, presents challenges that arise from potential incompatibilities among the respective components. Self-reinforcing "vicious cycles" caused by interactions between them lead to unstable or poorly-tuned adaptive components, consequently resulting in performance deterioration even in the absence of failures and bottlenecks. The problem essentially lies in *performance composability* or lack thereof; a challenge that arises because individual optimizations in performance-sensitive systems generally do not compose well when combined. Performance adaptation in such systems needs to be carefully designed and implemented by holistically considering the resulting combined behavior of adaptive components in order to achieve desired system performance. In this dissertation, we aim to develop (i) a software service to ease the implementation of holistic performance management techniques for controlling and optimizing system performance and (ii) runtime diagnosis and recovery techniques for guarding the system from instability.

We first develop a software service, called *OptiTuner*, that allows easy implementation of different holistic performance management schemes based on theoretical concepts of constrained optimization and feedback control. Such performance management approaches share common characteristics that are exploited by OptiTuner to provide a flexible supporting software layer. With the provided abstractions, APIs, and various services, OptiTuner monitors the current performance and the resource availability in performance-sensitive systems and executes implemented holistic performance management techniques to achieve performance goals. Further, OptiTuner provides a integration-time mechanism, called *adaptation graph analysis*, for identifying potential incompatibilities between composed adaptive policies. Since it is not always the case that the entire software system is designed together, different independently-designed subsystems may cause adverse interactions. Simplifying the concept of the stability condition in control theory, OptiTuner can detect such interactions at system integration time by reasoning about potential side-effects of component composition.

Due to the high complexity of large-scale software systems, the possibility of potential performance problems still remains at run-time. Undocumented or dormant behavior that was ignored by the designer may be triggered in some corner cases. This makes a seemingly correct model of system behavior assumed at design-time violated, consequently producing unexpected interactions between components. Software configuration changes even due to regular software updates may also invalidate the assumed system model causing performance problems. The purpose of our online diagnosis and recovery service, *AdaptGuard*, is simple: in the absence of an a priori model of the software system, anticipate system instability by identifying the "vicious cycle" causing the performance problem, and disconnect the problematic adaptive component, replacing it with conservative but stable open-loop control until further notice.

We demonstrate the benefits of the presented techniques by implementing realistic scenarios on a testbed comprised of 18 machines. Recognizing the growing concern for the energy problem in data centers, we take as a main scenario an energy minimization application in a three-tier Web sever farm with multiple energy saving policies. Other scenarios include dynamic memory control in a consolidate environment in response to the newly emergent cloud computing environments and a case study of a performance anomaly caused by unexpected interactions between an admission controller and the Linux anti-livelock mechanism. Empirical results obtained from such scenarios prove the efficacy of the approaches presented in this dissertation.

To my family in Korea

and my dearest Insun.

Acknowledgments

It is my great pleasure to thank those who made this thesis possible.

I would like to first express the deepest appreciation to my advisor, Tarek Abdelzaher, who has the attitude and the substance of a genius: he continually and convincingly conveyed a spirit of adventure in regard to research. Without his guidance and persistent help this dissertation would not have been possible.

Xiaoyun Zhu deserves a special thanks for her invaluable advice when I was an intern at HP Labs. The associated experience and work became an integral part of my thesis.

I would also like to thank my committee members, Indranil Gupta, and Matthew Ceasar for their enthusiastic and constructive comments and suggestions to improve the work.

I am indebted to my many of my colleagues to support me in collaboration with them: Xue Liu, Praveen Jayachandran, Maifi Mohammed Khan, Hieu Khac Le, Nam Pham, Pradeep Padala, Zhikui Wang, Insik Shin, Liqian Luo, and Dong Wang, and Lui Sha.

I would like to show my gratitude to all other members of the UIUC Cyber Physical Computing group and colleagues whom I have worked with: Marco Caccamo, Chengdu Huang, Qing Cao, Raghu Kiran Ganti, Yu-En Tsai, Yong Wang, Hossein Ahmadi, Yusuf Sarwar, Eunsoo Seo, Saurabh Nangia, Lin Gu, and Shen Li.

Last but not least, thanks be to my family members, my parents, grandmothers, brother, sister-in-law, uncles, and ants for their endless love through my entire life. My final, and most heartfelt, acknowledgment must go to my wife, Insun Yoon, who has always been there for me ever since we met. This thesis is dedicated to my family.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This work is to design, implement, and evaluate software support and design techniques for performance composition and dynamic performance management in performance-sensitive systems. As modern performance-sensitive systems become larger and their applications become more complicated, an increasing need arises for self-tuning and adaptive components to accommodate environmental dynamics and uncertainty [39]. This dissertation encompasses our efforts to develop (i) a software service to ease the implementation and composition of holistic performance management techniques for controlling and optimizing system performance and (ii) runtime diagnosis and recovery techniques for guarding the system from instability.

In this chapter, we first motivate the work by recognizing challenges in achieving good aggregate performance in large-scale performance-sensitive systems (Section 1.1). We then present our contributions (Section 1.2) and, finally, we outline the rest of the dissertation (Section 1.3).

1.1 Motivation and Challenges

The topic is motivated by the growing complexity of modern performance-sensitive systems, which increasingly calls for adaptive capabilities to reduce the need for manual tuning. Automatic energy management policies [21, 34], rate adaptation techniques (e.g., in 802.11 cards) [33], and congestion window adaptation (e.g., in TCP)[3] are common examples of adaptive policies in current systems. These recent trends are exemplified by IBM's autonomic computing initiative [39] that suggests increased need for self-managing, self-calibrating, self-healing, and self-tuning components in software systems. A proliferation of adaptive policies is thus expected in future software that automatically re-calibrate system parameters (e.g., fine-tune false alarm thresholds of sensors or change transmission power and frequency of wireless communication devices), re-distribute load (e.g., by load balancing), adapt the amount of resources available (e.g., turn off resources not in use), or control data flow topology (e.g., re-route around hot spots). These adaptive mechanisms will be executed unsupervised to reduce the need for human input.

While automatic adaptation can significantly reduce the need for human intervention (and hence software ownership cost), controlling and optimizing performance of large-scale adaptive systems are becoming a very hard problem. Subtle interactions between individually well-tuned components may result in unexpected and adverse emergent behavior contributing to significant performance degradation, due to a growing number of tunable parameters, abstractions that hide performance-related knobs, and limited observability of internal software state at run-time. Performance of the overall system becomes an emergent property that depends on the aggregate behavior of the individual adaptive components.

To better illustrate the problem of unintended adverse interactions among different performance adaptation modules [30, 31, 54, 55] in a large-scale performance-sensitive system, let us consider an energy minimization application in a 3-tier web server farm where the first tier provides a web interface to clients, the second tier implements business logic, and the third tier serves as a backend database. Two mechanisms are installed to save energy. The On/Off policy powers off some machines when the system is underutilized and powers on extra machines when overutilized. The DVS policy employs dynamic voltage/frequency scaling (DVFS) on individual processors such that the speed and voltage of a machine are adjusted based on the system load. However, when put together, their combined performance depends on the relative tuning of their respective overload and underutilization thresholds. If the DVS policy is too aggressive, whenever the utilization of a machine decreases, the policy reduces clock frequency (and/or voltage) thus restoring a high utilization value. From the perspective of the On/Off policy, the farm becomes “fully utilized”, as the measured utilization remains high. This will drive the On/Off policy to needlessly turn machines on in an attempt to relieve the full utilization condition. DVS will slow down the clock further, causing more machines to be turned on, and so on. This interaction may, in fact, increase the total energy consumption as the extra energy overhead of keeping more machines up (even at a low frequency) may dominate.

The above example of bad interactions between adaptive components are not uncommon and many similar problems are reported [28, 30, 54, 55]. The second example is from a Web Sphere server farm experiment where a load balancer distributed client sessions among servers equipped with a DVS capability. The load balancer attempts to balance response times. Occasionally, temporary utilization imbalance among servers caused by session load fluctuations caused the DVS policy on a lightly-loaded server to reduce CPU frequency, hence increasing delay. In response to such increased delay, the load balancer moved some work from the already lightly loaded server to other servers. The cycle repeated itself, as DVS took advantage of the reduced load and decreased frequency further, eventually leading to a very skewed load distribution.

A third example comes from the anti-receive-livelock mechanism in Linux [56]. In a server with a utilization controller running on a Linux system, when request rate increased beyond some point, the anti-livelock mechanism would switch from event-based to polling-based network I/O. This would result in reduction in utilization, to which the utilization controller would respond by taking action that increased request rate (e.g., tell a load balancer that the server can accept more requests). The increased rate only exacerbated the problem, ultimately resulting in a very high connection drop ratio.

In the above examples, no components failed and the degraded performance (in terms of energy consumption) was not caused by bottlenecks. The performance problems were in fact caused by self-reinforcing interactions between adaptive components that lead to bad states [30, 31, 54, 55]. In process control, such interactions are called feedback loop instability. A similar effect is also observed in software systems when component interactions produce poor resource allocation in the system reinforcing themselves repeatedly. Due to the unstable cyclic nature of these interactions, gradual performance degradation ultimately leads to highly suboptimal behavior. Such challenge in achieving good aggregate performance arises essentially because individual optimizations in performance-sensitive systems generally do not compose well when combined due to possible undesired interaction between them. The key in developing performance-sensitive systems is therefore that, *performance composability*, or the ability of individually well-optimized components to compose in ways that achieve good aggregate behavior, should be considered carefully by properly coordinating their combined effect and resource allocation. This observation is likely to be more true in large-scale distributed systems that have multiple ”knobs” for adjusting performance and resource allocation.

Due to the high complexity of large-scale software systems, the possibility of potential performance problems still remains at run-time even when software systems are optimally designed and composed in a way that reduces malicious interactions among components. To optimize and control the aggregate system performance, the design of adaptive components should make assumptions about their effect on resource allocation, other system components, and the external environment. Those seemingly correct assumptions, however, may be violated as the software configuration changes or undocumented or dormant behavior that was ignored by the designer is triggered in some corner cases. For instance, when a particular initial load condition is met, the secondary effect may become dominant actively interacting with other components. When assumption violations occur, the once coordinated behavior may respond inappropriately, consequently driving performance in the wrong direction in ways not predicted by the designer. Unfortunately, when instability caused by interactions among individually well-behaved components is the problem, it often can not be easily localized at run-time. There are no malfunctioning components to identify and no bottleneck nodes to isolate in order to explain the performance problem. To effectively detect and recover from the problem at run-time, therefore, it is critical to identify the "vicious cycles" or unstable closed-loop behavior causing the problem. It is also critical to accomplish this preferably without an explicit model describing the system, since it is usual that the exact system model is not known at run-time.

1.2 Contributions

To address the aforementioned challenges, a supporting software layer needs to be developed to facilitate holistic performance management techniques such that it is possible to compose adaptive components in a performance-maximizing fashion, control their resulting combined behavior, optimize run-time system performance autonomically in the face of unpredictability and avoid undesirable interactions. Moreover, since software is usually not designed from scratch but rather composed from new and legacy components, mechanisms are needed to check for composability at integration time and diagnose and recover from residual errors at run-time.

In this work, two directions are explored to provide (i) a software service to ease the implementation and composition of performance management techniques for preventing unwanted interactions, thereby achieving good aggregate performance and (ii) online diagnosis and recovery techniques at run-time for guarding the system from instability caused by emergent anomalous interactions. For specificity, these directions will focus primarily on interactions between adaptive components of distributed systems; a choice inspired by the increased need for future software system autonomy and hence for correct adaptive behavior. The benefits of the presented techniques are demonstrated and evaluated by implementing various realistic scenarios on a testbed comprised of 18 machines.

1.2.1 Software Service for Performance Composability

The next generation of performance-sensitive systems is expected to be more distributed and dynamic. They will have multiple "knobs" that affect performance and resource allocation. However, relying on the conglomeration of independent knob controls can become increasingly suboptimal.

To address the challenge of performance composability, performance adaptation needs to be carefully designed and implemented by holistically considering performance composability in order to achieve desired system performance. Various holistic design techniques for performance management such as optimization techniques and feedback control approaches can be considered to ensure proper adaptation to dynamic changes in the external environment or in system load that require manipulating performance knobs. A flexible supporting software layer is therefore required to register the available performance tuning mechanisms (called control knobs) in a large system and coordinate their operation to achieve or approach stated performance goals subject to applicable constraints. In this dissertation, we present *OptiTuner*, a software service designed to achieve the above.

The main contribution of our software service lies in facilitating the implementation of different holistic performance management mechanisms in large software systems based on principles of constrained optimization and control. OptiTuner provides simple abstractions and necessary services to support common operations of various performance optimization and feedback control techniques. With the supported abstractions and services, OptiTuner then runs a collection of performance management modules connecting performance monitoring and control knob manipulation modules. All modules in OptiTuner are pluggable and can be distributed, allowing flexible development for performance-sensitive systems. We explain our software service for performance composition, OptiTuner, in Chapter 2.

To thoroughly evaluate the efficacy of OptiTuner, we present two case studies in large-scale distributed systems. First, in response to the newly emergent cloud computing environments, a case study of dynamic memory control in a consolidated environment is presented in Chapter 3. Implementing such a system requires a flexible software support to easily access various kinds of sensors and actuators exported by the virtualization layer and the application. Using OptiTuner's services and APIs, we develop a joint dynamic CPU and memory controller based on the observation that memory allocation conflict with other components only in a certain corner case. By avoiding such a region, we show that the joint controller implemented using OptiTuner can achieve the desired performance goal.

In many cases, however, interactions between composed adaptive components are inherent and may not be completely avoided. The best solution in such cases is to proactively coordinate their combined behavior to reduce anomalous interactions, hence achieving the desired performance goals. In Chapter 4, we present an energy minimization application in a three-tier Web server farm comprised of 18 machines, where three types of energy saving knobs are used incurring constant interactions between them. In order to effectively minimize power consumption, those knobs need to be continuously adjusted and coordinated in the presence of workload changes. We implement three different holistic approaches for adjusting energy saving knobs and show that how OptiTuner can be used to control the combined behavior of those knobs, hence successfully achieving energy savings.

1.2.2 Online Diagnosis and Recovery

Assuming that models used to design adaptive components are not perfect, it may be that some unintended interactions survive composition-time checks and manifest themselves at run-time, creating system instability, and hence performance problems. However, performance problems resulting from such bad interactions often cannot be easily diagnosed by previous debugging approaches geared for detecting single component failures [11, 80], or those geared for isolating performance bottlenecks [2, 13, 43]. The key in recognizing such performance problems is to identify "vicious cycles" that can potentially explain the problem.

In this dissertation, we present *AdaptGuard*, an online diagnosis and recovery service for performance-sensitive systems that guards the target system from instability without the benefit of a priori system models in Chapter 5. The purpose of *AdaptGuard* is simple: in the absence of an a priori model of the adaptive software system, anticipate system instability, attribute it correctly to the right "runaway" adaptive component, and disconnect it, replacing it with conservative but stable open-loop control until further notice. Borrowing the concept of loop stability from classical control theory, it understands application-independent preconditions of instability, thereby correctly identifying the responsible adaptation loop (i.e. adaptive component) consistent with the "vicious cycle". While control theory can identify unstable adaptation loops when the variables involved can be described by difference or differential equations, the detection process by AdaptGuard is performed without knowing the system model a priori.

AdaptGuard provides two different diagnosis mechanisms. The first mechanism uses a simple statistical correlation analysis to *automatically* infer the implicit assumptions a designer must have made that have a bearing on stability. When they are violated, it expects instability to occur and performs intervention, breaking the actual runaway adaptive components. Since the first mechanism is more effective when variables are continuous, AdaptGuard provides another diagnosis technique that extends the first mechanism to detect discrete chains of events causing performance problems by using data mining techniques. It leverages discriminative sequence mining using logs of correct past behavior in order to identify, by contrast, any recent anomalous chains of events that are candidates for blame for the performance problem.

Our online diagnosis and recovery service, AdaptGuard, is evaluated by injecting various software faults into a QoS-adaptive Web server installed in the testbed and by using an energy minimization application in a three-tier server farm. A case study of a performance anomaly caused by unexpected interactions between an admission controller and the Linux anti-livelock mechanism is also presented to demonstrate the efficacy of AdaptGuard.

1.3 Outline

The rest of the dissertation is organized as follows. Chapter 2 presents a software service for performance composition, OptiTuner. In the following two chapters, we describe how OptiTuner can be used to develop performance management in different scenarios in distributed performance-sensitive systems: Chapter 3 describes a case study where OptiTuner is used for dynamic memory allocation to Xen virtual machines in a consolidated environment and Chapter 4 implements and evaluates various holistic performance management techniques using OptiTuner in an energy minimization application for a multi-tier Web server farm. Chapter 5 presents and evaluates our online recovery and diagnosis approach, AdaptGuard. Chapter 6 presents an extensive literature survey and, finally, Chapter 7 concludes the dissertation.

Chapter 2

OptiTuner: Software Service for Performance Composition

2.1 Introduction

This chapter develops a software service for dynamic performance optimization and control in performance-sensitive systems. The topic is motivated by the growing importance of adaptive capabilities to automatically manage and control system performance in large-scale software systems [3, 21, 33, 34]. These recent trends are exemplified by IBM's autonomic computing initiative [39] that suggest increased use of self-managing, self-calibrating, self-healing, and self-tuning components in various areas of future software.

Although such adaptive capabilities has considerably diminished the need for human intervention, controlling and optimizing performance of large-scale performance-sensitive systems are becoming a very hard problem. Subtle interactions between individually well-tuned components may result in adverse emergent behavior when combined contributing to significant performance degradation [30, 31, 54, 55]. Such incompatibilities may exist even though the adaptation mechanisms in question attempt to optimize the same performance objective. For instance, in the previous chapter, we presented an example of bad interactions between two energy saving policies in a three-tier Web server farm, where the interactions of the two policies eventually caused the server farm to use more machines than needed, hence wasting energy.

Such a performance problem is caused by self-reinforcing interactions between adaptive components that lead to bad states [30, 31, 54, 55]. Due to the unstable cyclic nature of these interactions, gradual performance degradation ultimately leads to highly suboptimal behavior. To address those problem, when designing adaptive components in performance-sensitive systems, *performance composability*, or the ability of individually well-optimized components to compose in ways that achieve good aggregate behavior, should be considered carefully. Various design techniques for performance management can be considered to ensure proper adaptation to dynamic changes in the external environment or in system load that require manipulating performance knobs. For example, optimization approaches [10, 12, 15, 30, 38, 50] can be used to find a best combination of the performance knob settings that optimizes performance, hence reducing undesired interactions among them. Similarly, feedback control approaches [17, 32, 78, 79] can be used to achieve desired performance set points by adjusting performance knobs properly. Further, a certain type of approaches may be preferred to others based on system requirements (e.g., distributed vs centralized). A flexible supporting software layer is therefore required to register the available performance tuning mechanisms (called control knobs) in a large system and coordinate their operation to achieve or approach stated performance goals subject to applicable constraints. In this chapter, we describe *OptiTuner*, a software service designed to achieve the above.

OptiTuner provides simple abstractions and necessary services to support common operations of various performance optimization and feedback control techniques such as constantly monitoring the current performance and resource availability of the target system and dynamically adjusting performance control knobs. OptiTuner then runs a collection of performance management modules connecting performance monitoring and control knob manipulation modules. All modules are pluggable and can be distributed, allowing flexible development for performance-sensitive systems.

Further, since it is not always the case that the entire software system is designed together, an integration-time mechanism is needed for identifying potential incompatibilities between composed adaptive policies. In OptiTuner, the user can register any integration-time checking module. As an example of such integration-time checking mechanisms, we present a simple heuristic called *adaptation graph analysis*. Inspired by control theory, the adaptation graph analysis analyzes interactions between independent components by reasoning about potential side-effects of component composition.

A running case study illustrates the usefulness of OptiTuner in this chapter. We present an autonomous energy minimization application in a multi-tier server farm testbed of 18 machines, in which two adaptive policies are used to save energy during off-peak load conditions. We show how unexpected adverse interactions may happen and make it non-trivial to integrate these adaptive policies into one coherent control framework. Using the adaptation graph analysis, OptiTuner identifies potentially adverse interactions. The technique uncovers the composition problem between the two policies. Further, it is shown that a holistic energy minimization approach implemented using OptiTuner indeed achieves significant energy savings while energy inefficiencies are clearly shown when the two policies are composed without regard to their interactions.

The rest of the chapter is organized as follows. Section 2.2 describes the design of OptiTuner. An integration-time checking mechanism, the adaptation graph analysis, for identifying potentially adverse interactions between adaptation policies is described in Section 2.3. Section 2.4 discuss the usefulness of OptiTuner by showing how it is used to analyze potential conflicts in the energy minimization application on our experimental testbed. Further, empirical results demonstrate that an holistic energy minimization approach implemented with OptiTuner considerably improves performance in terms of energy. Finally, conclusions are given in Section 2.5.

2.2 Design

A significant number of QoS adaptation and other performance adaptation policies have been reported in real-time computing literature over the last decade (e.g., [61, 66, 67]). In much of the current literature, these policies are designed and evaluated in isolation, showing efficacy in achieving the performance requirements of the system. However, unintended interactions between independently designed adaptive policies have not traditionally been addressed. As these policies gain popularity in deployed systems, a significant number of applications, middleware components and operating system mechanisms will exhibit adaptive behavior. A performance-sensitive software system of the future will therefore likely include multiple adaptive components. While these components will perform well in isolation, the interactions between them must be well coordinated to prevent unintended consequences.

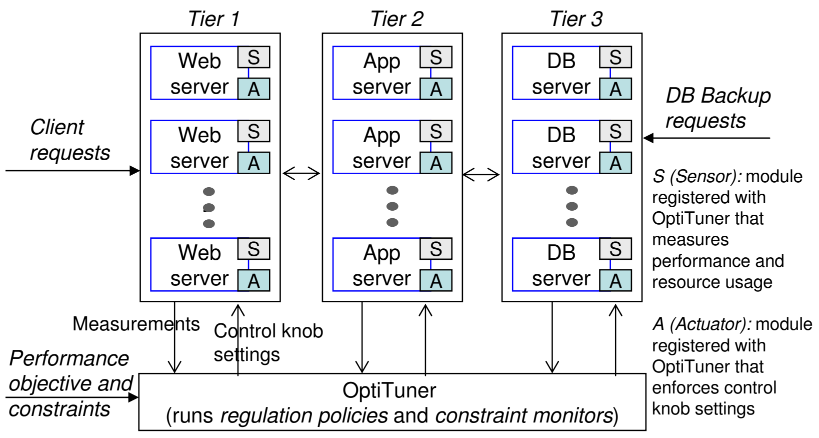

Tier 1
Tier 2
Tier 3
Web server
App server
DB server
S
A
Client requests
DB Backup requests
S (Sensor): module registered with OptiTuner that measures performance and resource usage
A (Actuator): module registered with OptiTuner that enforces control knob settings
Measurements
Control knob settings
Performance objective and constraints
OptiTuner
(runs regulation policies and constraint monitors)

Figure 2.1: An energy minimization application in a 3-tier Web server farm with OptiTuner.

As a main example to illustrate how OptiTuner can be used for designing and implementing performance-sensitive systems, we develop an energy cost minimization application for 3-tier Web server farms and use it throughout this dissertation. We assume that a server farm runs a typical 3-tier Web application and provides a database backup service for other organizations to fully utilize its extra computing power (e.g., CPU and storage). The total energy cost incurred by the system is composed of two parts: (i) the energy cost of server machines in the system and (ii) less the utility achieved from database backup requests. The objective of the optimization problem is to minimize the total cost incurred by the system, subject to the delay constraints of client requests and a resource constraint on the number of available servers.

Three different types of performance control knobs can be used to adjust power consumption. First, we adjust the number of assigned machines for the server farm by dynamically turning machines on and off. Second, dynamic voltage frequency scaling on an individual processor is used to change the speed and voltage of the processors in an active machine. It is reduced when the machine is underloaded and increased when it is overloaded. Finally, we adjust the rate of the incoming database backup requests for the database tier using admission control. We call the three types of performance control knobs On/Off, DVS, and BackupAC respectively.

In order to effectively minimize desired power consumption, energy saving knobs need to be continuously adjusted and coordinated in the presence of workload changes. A certain holistic technique for performance adaptation needs to be employed to coordinate performance knobs to achieve the performance goal (e.g., minimizing power consumption) while meeting the imposed constraints (e.g., end-to-end delay and the number of available machines).

Focusing on the critical issue of coordination of different knob settings in performance-sensitive systems, OptiTuner provides a supporting software layer for composing performance adaptation modules in ways that follow sound theoretical concepts of performance optimization and control. The contribution lies in the ease of applying different holistic optimization and control techniques simply by changing the policy modules that sit between performance measurement and control interfaces.

In general, performance adaptation modules (i.e. adaptive components) are at their very essence feedback loops (i.e. adaptation loops) as depicted in Figure 2.2. These loops take measurements of the system (e.g., request rate, delay, utilization, or queue length) and respond by a corresponding action that adjusts some variable such as the amount of resources allocated to a given process or the priority of a thread. Such common structure observed in different performance adaptation design techniques leads to a set of

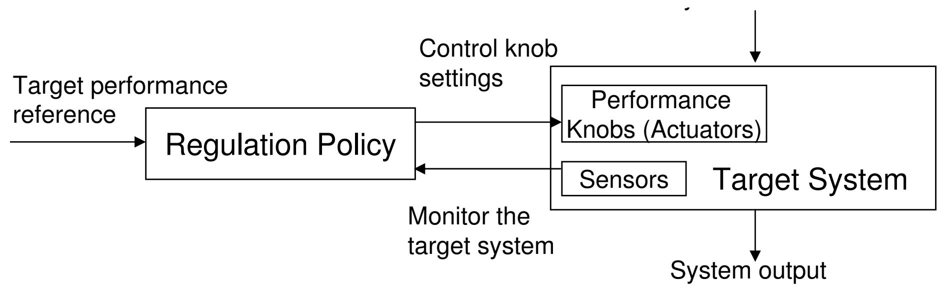

Target performance reference
Regulation Policy
Control knob settings
Performance Knobs (Actuators)
Sensors
Target System
Monitor the target system
System output

Figure 2.2: Performance-adaptive system architecture

abstractions exported by OptiTuner. In OptiTuner, performance control knobs are governed by *regulation policies*. Regulation policies in OptiTuner are essentially performance adaptation modules periodically determining the values of the performance control knobs. In order to help the decision of regulation policies effectively, OptiTuner exports *performance sensors* that are thin software interfaces wrapped around pertinent performance measurements. *Actuators* are software modules that enforce the settings of the control knobs determined by regulation policies. *Constraint monitors* monitor the availability of resources to inform individual regulation policies of how much further performance can be improved without breaking the constraints.

These abstractions are implemented as corresponding objects registered with OptiTuner. Since OptiTuner runs on all machines in the target system to properly monitor and act on it, *nodes* are introduced to abstract the involved physical machines. The purpose is to simplify the communication between the objects on different machines and manage them efficiently. Using the exported abstractions and the provided object access mechanisms, OptiTuner runs a collection of regulation policies that connect to the right registered sensors and actuators. By changing the policy, a system administrator can experiment with different holistic optimization and control schemes.

In the next section, we first present the system architecture and services provided to support the execution and management of object abstractions. We then explain APIs for implementing holistic performance management techniques based on performance optimization and control.

2.2.1 System Architecture

An OptiTuner process runs on each machine involved in the target performance-sensitive system to properly capture the system state. All modules in OptiTuner are distributed objects and can reside on any machine.

To support this, the OptiTuner process in each machine is essentially an object container that manages local objects. Further, every OptiTuner process in the target system is seen as a peer by others, providing communication methods for the objects running on other machines. This peer-to-peer property is desirable since OptiTuner needs to support performance management in large-scale distributed systems.

To provide an uniform access to objects across all OptiTuner processes, a global configuration file is shared among the OptiTuner processes. Simply put, the configuration file contains the information of the registered objects: which machine manages what objects and the properties of the objects. Object properties include sensing period, measurement value types, and an invoking period for regulation policies. Based on the configuration information, each OptiTuner process initiates itself by starting up the objects residing in the machine. While having the global view of the target system, each OptiTuner process focuses only on managing the local objects by providing necessary remote communication methods for the remote objects. With this architectural support, any regulation policies can easily find the location of any necessary sensors as well as their properties to connect to them.

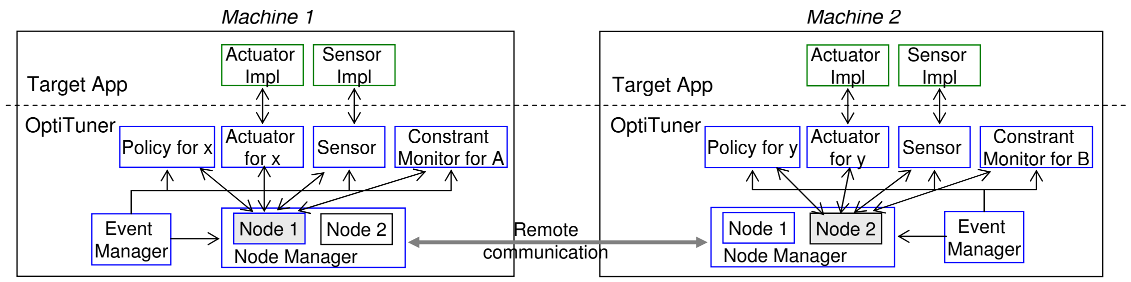


Figure 2.3: Runtime architecture of a performance-sensitive system with OptiTuner

The *node manager* manages a list of node objects that represent all machines controlled by OptiTuner, while each node object maintains a table that caches local objects. In OptiTuner, objects that need periodic invocation register themselves with the *event manager*. The even manager then invokes the registered objects based on their execution period. It should be noted that most OptiTuner objects are periodic. For instance, regulation policy objects adjust their corresponding control knobs periodically and, similarly, sensor objects usually collect data periodically.

Our design of OptiTuner doesn't limit the choice of the implementation language. The current prototype implementation of OptiTuner is written in Python. Choosing a script language didn't affect the performance severely, because most of the computation is done periodically, hence consuming negligible CPU time. Our measurement shows that OptiTuner implementation written in Python consumes less than 3% of CPU utilization most of the time.

2.2.2 Object-Based APIs

OptiTuner provides simple object-based application programming interfaces (APIs) for application programmers to easily implement and run holistic performance management techniques.

The two most basic functionalities required for executing performance management algorithms is 1) to gather necessary measurement data from the target system (e.g. sensing) and 2) to enforce the decisions made by the performance management algorithms on the target system (e.g. actuation). For this purpose, it defines two remote methods in the node object class for accessing the sensor and actuator objects residing in remote nodes. With the object name and its location (the node name) that are specified in the configuration file, any sensor and actuator objects registered in OptiTuner can be uniformly accessed. Given the name of a sensor, *node.getSensorVal(sensor_name)* returns the value of the sensor. Actuators are invoked by calling *node.invoke(actuator_name, value)*. Observe that regulation policy and constraint monitor objects also need to export their corresponding knob settings and price values so that the exported values can be used by other regulation policy and constraint monitor objects. The exported values can be accessed by using the same method, *node.getSensorVal(sensor_name)*, since regulation policy and constraint monitor object classes are inherited from the sensor object class. Hence, using the two methods, application programmers can access any objects (and their exported values) to implement regulation policy and constraint monitor objects.

As explained, the objects that need a periodic invocation to do their jobs register themselves with the even manager by specifying their desired period. As the timer for an object expires, the event manager will call back the *tick()* method of the object. Hence, the *tick()* method of an object can be used as a placeholder for implementing periodic jobs such as collecting measurement data (for sensor objects), updating knob settings (for regulation policy object), and updating price values (for constraint monitors).

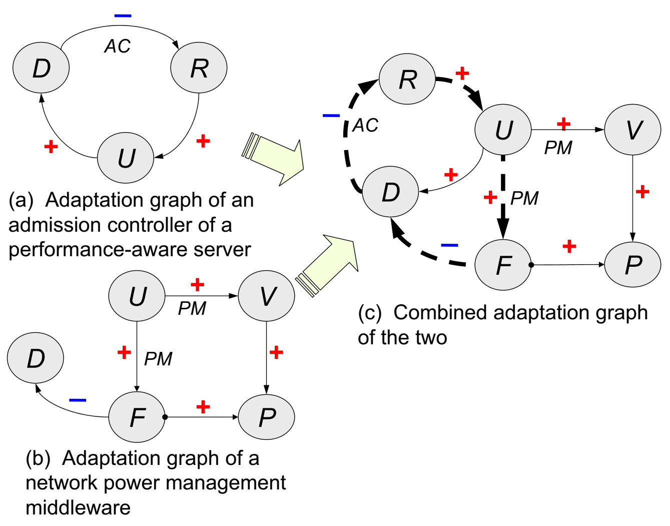
2.3 Adaptation Graph Analysis

Within each independently designed subsystem, the adaptation loop is typically well tuned. A problem occurs if an unintended loop emerges by virtue of composition of multiple subsystems. OptiTuner provides an interface with which the user can register and run an integration-time checking mechanism to analyze the behavior of combined adaptive components. In this section, we explain a simple heuristic approach called

Adaptation Graph Analysis. Inspired by control theory, the adaptation graph analysis can detect potential instabilities caused by conflicting adaptive components at system integration time.

2.3.1 Adaptation Graphs

To uncover unintended loops, we present the notion of adaptation graphs. Nodes in an adaptation graph represent the key variables in the system such as delay, throughput, utilization, length of different queues, settings of different policy knobs, etc. Arcs represent the direction of causality. For example, consider a Web server that serves requested pages over a network. When the utilization, U, of the outgoing link (connecting the server to the Internet) increases, the delay, D, of served requests increases as well (because they wait longer to be sent over the congested link). Hence, an arc exists from utilization to delay, $U \rightarrow D$, indicating that changes in the former affect the latter. The arcs in the adaptation graph are annotated by either a "+" or a "$-$" sign depending on whether the changes are in the same direction or not. In our above example, since an increase in utilization causes a *same-direction* change in delay (i.e., also an increase), the arc is annotated with a "+" sign: $U \rightarrow^{+} D$. Moreover, some of the arcs represent fundamental natural phenomena (for example, an increase in delay is a natural consequence of an increase in utilization). Others, however, represent *programmed* behavior, or policies. For example, an admission controller of a performance-aware server may be programmed to decrease the fraction of admitted requests, R, to the server in response to an increase in delay, D. Hence, an arc exists in the adaptation graph from delay to admitted requests, $D \rightarrow^{-} R$. The arc is annotated with a "$-$" sign because an increase in delay results in a change in the opposite direction (i.e., a decrease) in admitted requests. This arc does not represent a natural phenomenon but rather the way the admission control policy is programmed. We call such arcs *policy arcs* and annotate them with the name of the module implementing the corresponding policy. Hence, we have $D \rightarrow^{-}_{AC} R$, where AC stands for the admission control module. Figure 2.4(a) depicts the adaptation graph of the server under consideration. The graph is composed of three arcs. The arc $D \rightarrow^{-}_{AC} R$ reflects that the admission controller reduces the number of admitted requests when delay increases and vice versa. The arc $R \rightarrow^{+} U$ reflects the natural phenomenon that any changes in the number of admitted requests result in same-direction changes in outgoing link utilization. Finally, the arc $U \rightarrow^{+} D$ expresses the fact that changes in link utilization cause changes in delay (in the same direction). The three arcs form a cycle (a feedback loop). An interesting property of the loop is that the product of the signs of the arcs is negative.



(a) Adaptation graph of an admission controller of a performance-aware server

(b) Adaptation graph of a network power management middleware

(c) Combined adaptation graph of the two

Figure 2.4: Examples of adaptation graphs

This indicates a negative feedback loop, which is expected (since all stable feedback loops are negative).

As another example, consider a network power management middleware that measures network utilization, U, on the LAN of a load-balanced server cluster. If the network utilization is low, the cluster workload must be low. The middleware thus engages dynamic voltage scaling (DVS) on all machines in the cluster to lower processor voltage V, and frequency, F, hence reducing power consumption, P, due to the off-peak load condition. This adaptation action can be expressed as $U \to_{PM}^{+} V$ and $U \to_{PM}^{+} F$, where PM stands for power management middleware (i.e., a decrease in link utilization causes the policy to decrease both voltage and frequency which explains the signs on the arcs). In turn, we have $V \to^{+} P$ and $F \to^{+} P$, which says how power consumption changes with voltage and frequency. Finally, we have $F \to^{-} D$, since lowering frequency (i.e., slowing down a processor) increases delay and vice versa. Figure 2.4(b) depicts the adaptation graph for the network power management middleware.

As might be inferred from above, each component or subsystem of a larger system has its own adaptation graph that describes what performance variables this component is affecting and what causality chains (or loops) exist within. The reader might notice that the adaptation graph is a simplified version of the block diagrams used in control theory to represent open loop and closed loop control systems. We do not use full-fledged block diagrams (complete with transfer functions of components) because transfer functions require more accurate component modeling and present analysis challenges in the presence of non-linearities, as would be common in computing systems.

When a system is composed, the adaptation graphs of individual components are coalesced. Fig. 2.4(c) shows the combined adaptation graph that results when a server described in Fig. 2.4(a) operates on a LAN cluster managed by the middleware described in Fig. 2.4(b). This creates a system-wide graph. To check for incompatibilities (adverse interactions), the graph is searched for loops using any common graph traversal algorithm. Loops that are entirely contained within the same subsystem (i.e., loops whose arcs are labeled by the same module name) are safe. Such loops are internal to single components and must have been well-tuned prior to composition. Loops that traverse component boundaries, however, may have not been created by design. These loops (i.e., loops where some arcs have different module names) may have emerged unintentionally due to composition of the corresponding modules. This insight leads to simple techniques (carried out at integration time) to discover potentially unintended interactions as described below.

2.3.2 Checks for Potential Incompatibility

We suggest two simple checks for potentially incompatible adaptation policies. These checks operate on the adaptation graph of the composed system as follows:

- *Positive feedback:* A key requirement of adaptation graphs is that all cycles in the graph must be of a negative sign (i.e., the product of all arc signs on the cycle is negative). This requirement stems directly (and can be easily proved) from stability conditions in control theory [9]. A positive cycle indicates a potentially unsafe feedback loop. In other words, a stimulus reinforces itself causing more change in the same direction. Such a cycle may inadvertently develop when multiple adaptation policies are combined. All positive feedback loops in the adaptation graph are thus flagged as potentially unintended interactions.

- *Unstable negative feedback:* Another problematic condition that can be flagged when composing adaptation graphs is potentially unstable negative feedback loops. While adaptation loops that are governed by a single module will tend to be well-tuned and hence stable, emergent adaptation loops that arise from a combination of multiple modules must be explicitly analyzed for stability. All loops where some arcs have different module labels in the adaptation graph are thus flagged as potentially unintended interactions. Another example of potentially unstable negative feedback is if loops from different modules join in a node, indicating that the loops may interfere while trying to control the common variable. The web server case study is an example of this situation. Again, even if the individual loops are well-tuned and stable, they may interact in a way that degrades total performance or even causes instability.

Applying the above checks to Fig. 2.4(c), we flag the cycle $U \rightarrow^{+}_{PM} F$, $F \rightarrow^{-} D$, $D \rightarrow^{-}_{AC} R$, $R \rightarrow^{+} U$ that crosses module boundaries, suggesting it may have not been created by design. The cycle is also positive indicating that unstable interactions may result (between the server admission controller and the network middleware). Interpreting this cycle, the interaction is explained as follows. Starting with the node labeled, U, when the network utilization decreases in the server cluster, the power management middleware will cause individual servers to slow down their processors. This, in turn, will increase the delay experienced by served requests causing the admission controller to accept fewer requests. The reduced accepted number of requests will further decrease the load on the network, causing the power management middleware to slow down processors even more. This, in turn, will cause a more significant reduction in admitted requests and a further reduction in network load. This positive (i.e., self-reinforcing) feedback cycle will ultimately bring the server farm to a crawl, indeed an adverse consequence of unintended interaction.

2.4 Evaluation

In this section, we demonstrate the efficacy of OptiTuner using the energy minimization application for a three-tier server farm installed with two different types of energy saving policies (DVS and On/Off).

We first show that composing two energy saving policies (DVS and On/Off) without considering their combined behavior can lead to an excessive energy spending problem using OptiTuner's adaptation graph analysis. Let us now analyze the undesirable interaction between a DVS policy (that controls frequency, f, of machines in a server farm given their delay D) and an independently designed machine On/Off policy (that increases the number of machines m in the server farm when the delay is increased and removes machines when the delay is decreased). This interaction is shown in Fig. 2.5. The DVS policy is described by the adaptation rule $D \rightarrow^{+}_{DVS} f$. Its effect is $f \rightarrow^{-} D$. The On/Off policy is described by the rule $D \rightarrow^{+}_{ON/OFF} m$. Its effect is $m \rightarrow^{-} D$. The adaptation graph for the combined system is shown in Fig 2.5 (c), in which we see that the loops representing the two policies join in the common D node. This indicates a possible incompatibility, which may lead to performance degradation or even unstable behavior when the two policies are combined.

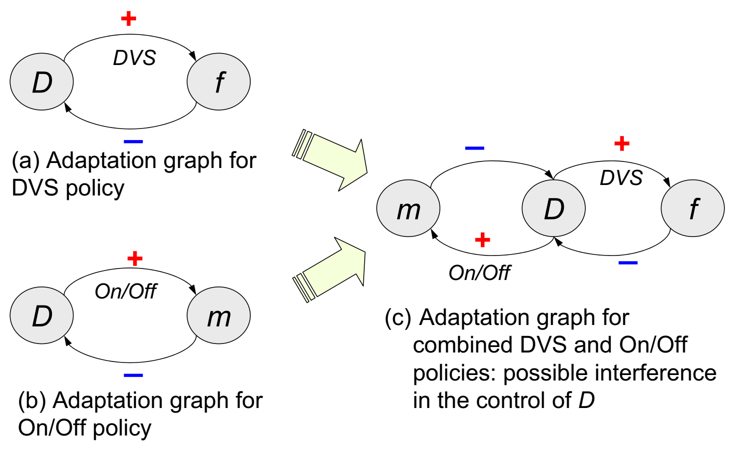

+
DVS
D
f
−
(a) Adaptation graph for DVS policy
+
On/Off
D
m
−
(b) Adaptation graph for On/Off policy
−
+
DVS
m
D
f
+
On/Off
−
(c) Adaptation graph for combined DVS and On/Off policies: possible interference in the control of D

Figure 2.5: Adaptation graphs for the different adaptive policies for the multi-tier server farm.

Indeed, as flagged by the analysis, the two individually stable negative feedback loops actually exhibit an undesirable interaction that may increase total energy consumption instead of reducing it. The adverse interaction works as follows. When the system is underloaded, the DVS policy reduces the frequency of a processor, increasing system utilization. This will eventually increase the end-to-end delay of the system. Increased delay may cause the (DVS-oblivious) On/Off policy to consider the system to be overloaded, hence turning more machines on to cope with the problem. This may cause the DVS policy to slow down the machines further, leading the On/Off policy to turn more of them on. The energy expended on keeping a larger number of machines on may not necessarily be offset by DVS savings. Hence, the resulting cycle may lead to poor energy performance, even despite the fact that both the DVS and On/Off policies have the same energy saving goal.

Fig. 2.6 shows experimental results from our three-tier Web server farm testbed. An energy minimization application is implemented in the testbed using OptiTuner. The detailed setup and complete experimental results are described in Chapter 4. In this chapter, we present results of a set of experiments where four different energy saving configurations are compared: the On/Off policy, the DVS policy, the combination of On/Off + DVS (exhibiting adverse interaction) and finally a holistic energy saving approach where the two knobs are holistically adjusted (we will discuss the details about various holistic approaches in Chapter 4). In the figure, it is clearly demonstrated that when the workload increases, the combined On/Off + DVS policy spends much more energy than all other policies. In contrast, the holistic policy shows best performance, especially for high workload. This experiment, performed on an actual server farm, indicates that

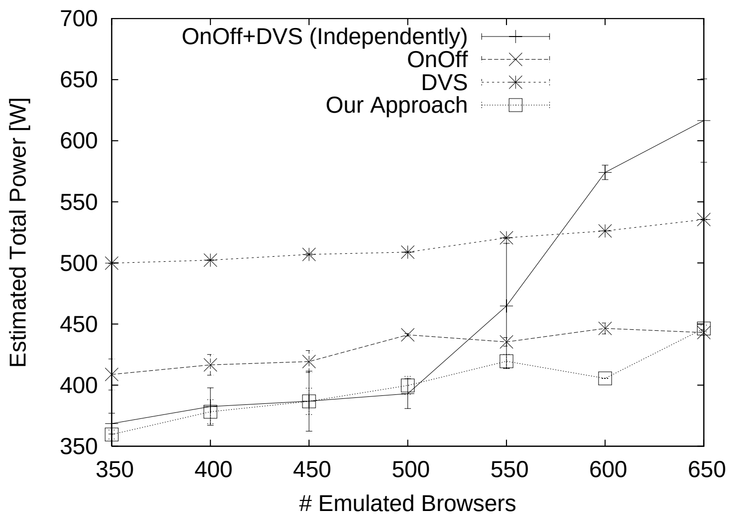


Figure 2.6: Comparison of estimated total system power consumption for different adaptive policies.

the adverse interaction problem is not hypothetical but is something measurable in practice. Indeed, as the figure shows (at high loads), combining two good energy saving policies without considering performance composability may yield savings that are worse than with either policy in isolation. Instead, they should be properly coordinated by considering the combined result to improve overall performance.

In the next two chapters, we describe how OptiTuner can be used to develop performance management in different scenarios in distributed performance-sensitive systems. In Chapter 3, we develop a joint dynamic CPU and memory controller in a virtualized environment using Xen virtual machines. The controller utilizes the observation that memory and CPU allocations conflict each other only in a certain corner case. By avoiding such a region, we show that the joint controller implemented using OptiTuner can achieve the desired performance goal. This is different from the energy minimization application shown in this chapter where energy saving knobs constantly interact each other that require proactive coordination to reduce bad interactions between them.

Chapter 4 presents a detailed case study of the energy minimization application in a three-tier server farm discussed in this chapter. We develop three different holistic performance management techniques for energy minimization using OptiTuner. Thorough empirical results are presented to prove the usefulness of OptiTuner for aiding in implementing different holistic approaches.

2.5 Conclusions

In this chapter, we presented a software service, called *OptiTuner*, for achieving performance composability in distributed performance-sensitive systems. Achieving performance composability in such systems is a great challenge because individual optimizations in performance-sensitive systems generally do not compose well when combined due to unexpected and emergent interactions between them. OptiTuner provides proper abstractions and necessary services to help the implementation of performance management approaches for coordinating different knob settings. OptiTuner also provides a mechanism, called *adaptation graph analysis* to identify potential incompatibilities between multiple adaptation policies. This is useful when the administrator needs to integrate adaptive components into a legacy system. The efficacy of OptiTuner was briefly demonstrated by applying it to energy minimization in multi-tier server farms. Using adaptation graph analysis, we first identified incompatibilities between the *On/Off* and *dynamic voltage scaling* policies used in the server farm. Experimental results demonstrate that a holistic energy minimization approach implemented with OptiTuner can save a considerable amount of energy compared to the ones that do not holistically optimize energy within acceptable bounds.

Chapter 3

Performance Management In Virtualized Environment

3.1 Introduction

In the previous chapter, we presented a software service, *OptiTuner*, to aid in application of holistic performance management approaches. In this chapter, we evaluate OptiTuner thorough a case study where feedback control is implemented for joint dynamic memory and CPU allocations to Xen virtual machines in a consolidated environment. The case study is in part motivated by the newly emergent cloud computing environments. In developing such a joint dynamic memory and CPU control mechanism with OptiTuner, the key observation is to recognize that memory, disk I/O operations, and CPU badly interact with each other only when a certain condition is met in terms of application-level performance. Based on this observation, a joint CPU and memory allocation mechanism for virtual machines is designed and developed by not allowing the corner case where bad interactions between memory and other components happen, thereby achieving the desired performance goal.

The cloud computing environments such a such as Amazons EC2 [4] that host hundreds to thousands of services on a shared resource pool. The sharing is enhanced by virtualization technologies such as Xen [5] and VMware [72] allowing multiple services to run in different virtual machines (VMs) in a single physical node. The same technologies have been used by enterprises to consolidate servers in order to improve resource efficiency, reduce data center footprint, and to reduce power consumption and environmental impact of IT organizations. Although the average resource utilization in traditional data centers ranges between 5-20%, servers in consolidated data centers or cloud computing environments are likely to run at much higher utilization levels, exposing applications to possible resource shortages [73] .

Researchers in both academia and industry have studied mainly three resource management strategies for consolidated environments capacity planning [64] , virtual machine (VM) migration [37] , and dynamic resource allocation. These techniques are complementary to one another because they typically operate at different time scales and different scopes of a data center [86]. In this chapter, we focus on dynamic resource allocation in a virtualized server that hosts multiple services. Runtime allocation of server resources to individual VMs enables *resource overbooking*, where the total capacity of a resource is below the sum of the peak demands of all the VMs sharing this resource. This allows each physical server to achieve much higher resource utilization while still maintaining performance isolation among the co-hosted services.

In the past, researchers have mostly studied overbooking of the CPU resource on virtualized servers and dynamic control mechanisms for allocating CPU to the individual virtual machines [59, 81, 85]. At the same time, memory overbooking and runtime re-allocation of memory among multiple VMs has not been widely studied, except in the VMware ESX Server [74]. However, the dynamic memory management policies in ESX do not directly support application-level performance assurance. The availability of memory *balloon driver* in recent Xen Server releases [83] provides a mechanism for memory sharing, but no dynamic policies have been implemented.

We have built an experimental testbed using Xen virtualized servers and performed a set of experiments to understand the relationship between memory allocation to a VM and performance of the hosted application. We observe that this relationship is different from the relationship between CPU allocation and application performance. We further recognize that memory allocation may have conflicts with disk I/O and CPU allocation when memory utilization is over a certain threshold, thereby severely affecting application performance once fallen into that region. To meet the desired application-level performance goal, such "bad" region must be avoided.

Based on these observations, we make the following two contributions in this chapter. First, we have designed a dynamic memory allocation controller that ensures that each VM gets sufficient memory to avoid anomalous interactions between CPU and memory. We validate this controller against time-varying memory demands and demonstrate that memory overbooking on Xen can be achieved using our controller. Second, we have built a prototype of a joint CPU and memory control system for a consolidated environment. Implementing such a system requires a flexible software support to easily access various kinds of sensors and actuators exported by the virtualization layer and the application. The control system is implemented using OptiTuner's services and APIs to achieve the goal. We evaluate the performance of this control system by driving multiple test applications with either synthetic or real world demand traces, and demonstrate that all the hosted applications can achieve their service level objectives (SLOs) without creating CPU or memory bottlenecks.

3.2 System Setup and Architecture

Using OptiTuner, we have built a testbed for experimentation, modeling, development, and performance evaluation of dynamic memory allocation schemes in a consolidated environment. In this section, we first briefly describe the setup of our testbed, and then present the architecture of a feedback-based resource control system we developed.

3.2.1 Testbed Setup

Our experimental testbed consists of 4 HP Proliant DL 320 G4 servers. Each server has dual 3.2GHz Pentium D processors with 2MB L2 cache and 4GB main memory. The servers run SLES 10.1 or SLES 10.2 with a Xen-enabled 2.6.16 kernel. One server is used to run a control application that monitors and controls the resource sharing of the virtual machines. A second server is configured to have 1-4 production VMs, each running an Apache Web server (version 2.2.3). Each of the last two servers hosts two client VMs, where *httperf* (ftp://ftp.hpl.hp.com/pub/httperf), a scalable client workload generator, is used to send concurrent HTTP requests to the Apache Web servers running on the production VMs. The virtual machines run the same kernel image, SLES 10.1, and have the same /usr file system. The virtual machines and the servers are located on the same network, interconnected via a Gigabit Ethernet.

We have developed an application called *MemAccess* that drives the memory usage and CPU consumption of a VM from trace files. We employ an Apache module to implement *MemAccess* on the Apache Web server. When a new HTTP request arrives, this Apache module is invoked to execute some computation (e.g. public key encryption) to consume a certain amount of CPU time, while randomly accessing a portion of the heap memory of the *MemAccess* application.

Memory usage patterns in a trace file are recreated by properly adjusting the size of the allocated heap in the Apache module over time. The heap memory is created when the Apache process starts and it consists of memory chunks of 5MB each. Memory chunks are allocated or de-allocated based on the memory load in the trace. The heap is randomly accessed when requests are received as explained formerly, to actively utilize the entire heap region.

CPU consumption of the application is adjusted by properly changing the average rate of HTTP requests generated from an *httperf* client. We calibrated the CPU consumption per request in order to compute the needed request rate for a VM's CPU consumption to match the CPU consumption in the trace. The calibration process is performed in one VM, since all the VMs running the Apache server are configured the same.

3.2.2 Control System Architecture with OptiTuner

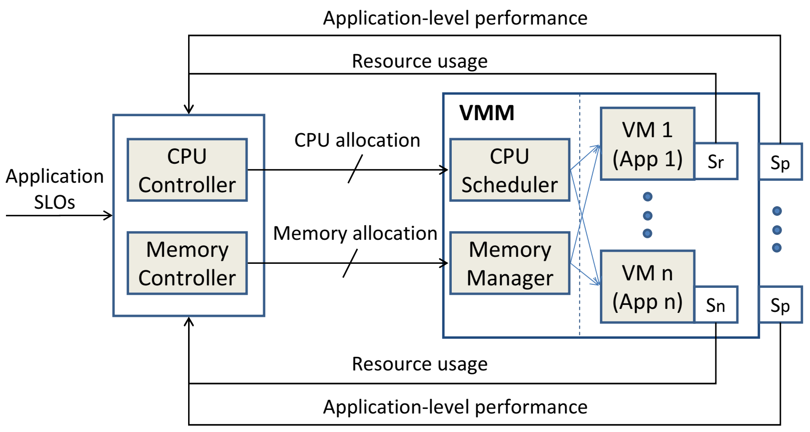


Figure 3.1: Runtime architecture of a performance-sensitive system

Figure 3.1 shows the architecture of our resource control system, where two or more applications running on virtual machines share the resources in a physical node. The resource control system is developed using our software service, OptiTuner, which proves that it is also useful for performance management in a virtualized environment. OptiTuner processes run on all physical machines (including the ones running the Xen virtual machine monitor) and VMs as well to allow access to sensors and actuators exported by them to monitor application performance and control resource allocations to VMs. All such sensors and actuators are encapsulated and registered with OptiTuner to run as objects. A dedicated machine is used for the resource controller running as a regulation policy in OptiTuner.

The Xen virtualization layer allow us to allocate portions of the CPU and the physical memory to VMs to achieve desired performance. The resource controllers gather information on the resource usage of VMs and application performance from a set of sensors, as well as the service-level objectives (SLOs) for the applications, and determines in real time the CPU and memory allocations for all the VMs in the next control interval. Details about the actuators, sensors and controller are described below.

A *CPUActuator* object is implemented to use the Xen credit scheduler for setting the CPU shares for

VMs. The credit scheduler provided by the virtualization layer allows each VM to be assigned a *cap*, which limits the amount of CPU a VM can use. This non-work-conserving mode of CPU scheduling allows better performance isolation among multiple VMs, preventing a poorly-behaving application from draining the CPU capacity.

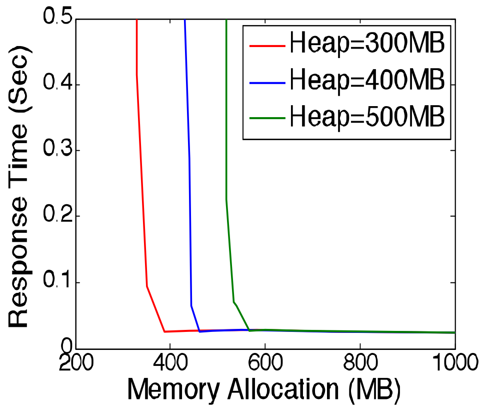
A *MemActuator* object uses the *balloon driver* in Xen to dynamically change the memory allocations for the virtual machines. Each VM is configured with a maximum entitled memory (*maxmem*). The value of *maxmem* is the sum of the balloon value and the allocation value. If one VM does not need all of its entitled memory, the unused memory can be transferred to another VM that needs it. This is done by inflating the balloon (reducing allocation) in the first VM and deflating the balloon (increasing allocation) in the second VM. This mechanism allows multiple VMs to be configured with a higher amount of total memory than the size of the physical memory.

Various sensor objects are implemented and configured to run in OptiTuner to periodically collect resource consumption and application performance statistics. CPU consumption is measured using Xen's xentop command. Memory allocation and usage are measured from the balloon in the /proc file system. We also monitor page fault rates of individual VMs using the /proc interface. Application performance is measured by an interposing proxy between the client and the server. Our resource controller consists of two parallel controllers for CPU and memory. Each controller periodically changes the CPU or the memory allocations for the individual VMs in every control interval. The control decisions are made based on a quantitative model inferred from black-box testing of the system, described in the next section.

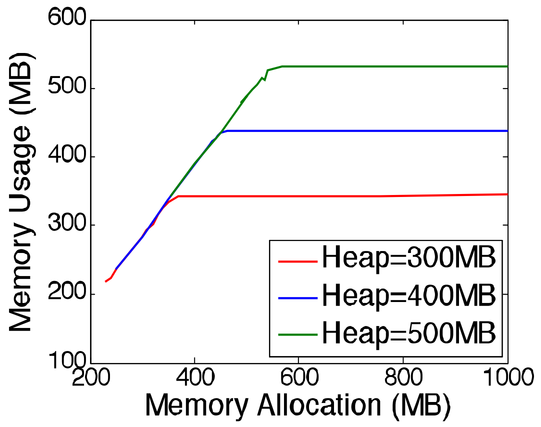
3.3 Memory Usage and Performance

In order to develop the memory controller shown in Figure 3.1, we need to first understand the quantitative relationship between an applications performance and its memory allocation. To this end, we performed the following experiments using our Xen-based testbed and the *MemAccess* application described in Section 3.2.

We ran the *MemAccess* application in one of the production VMs. An *httperf* client generated a workload with exponentially distributed inter-arrival times and a mean request rate of 30 requests/s. We allocated certain heap size for the application, and then varied the memory allocation to this VM from 230 MB to 1 GB. For each memory allocation setting, we ran the experiment for 50 seconds to allow the system to settle down to a steady state. In the meantime, the credit scheduler was running in the work-conserving mode so no CPU cap was imposed on the VM. The following metrics were collected during that 50-seconds interval: average memory usage, mean response time (MRT) of the application, and rate of major page faults. The experiment was repeated three times, for a different application heap size of 300, 400, and 500 MB, respectively.



(a)

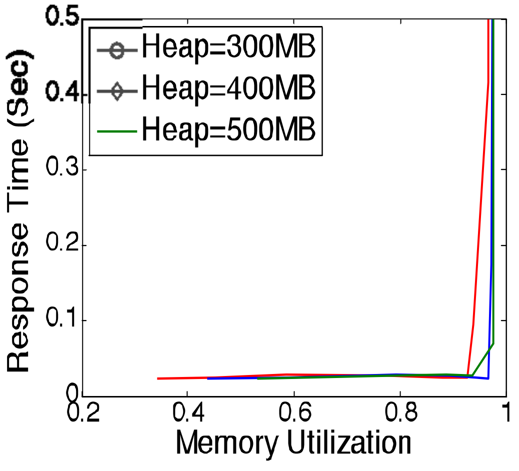


(b)

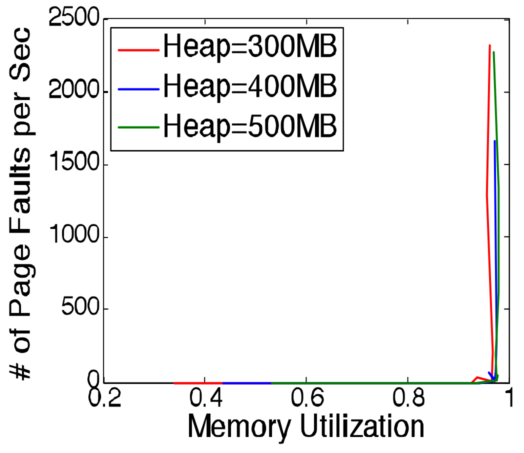
Figure 3.2: Mean response time (a) and memory usage (b) as a function of memory allocation

Figure 3.2(a) displays the measured MRT as a function of the memory allocation for the three different heap size configurations. For each configuration, the MRT of the application remains at around 20 ms for most values of the memory allocation until the latter drops below a certain threshold. To better understand this behavior, we also show the measured memory usage versus the memory allocation in Figure 3.2(b). We can see that the memory usage is constant when enough memory is allocated, and starts to decrease linearly with the memory allocation when the latter becomes small.

In Figure 3.3(a), we demonstrate the MRT as a function of the average memory utilization for the three configurations, where memory utilization of a VM is computed as the ratio of memory usage to memory allocation. It is easy to see that the MRT increases sharply as the memory utilization goes beyond 90%. This is consistent with our expectation, because when memory utilization is high, the guest operating system experiences significant memory pressure and starts reclaiming memory by paging a portion of the application memory to disk [4]. This will lead to a higher number of page faults when the application tries to access the main memory, resulting in higher latency for the application. To validate this, we also plot the number of major page faults per second as a function of the memory utilization in Figure 3.3(b). The



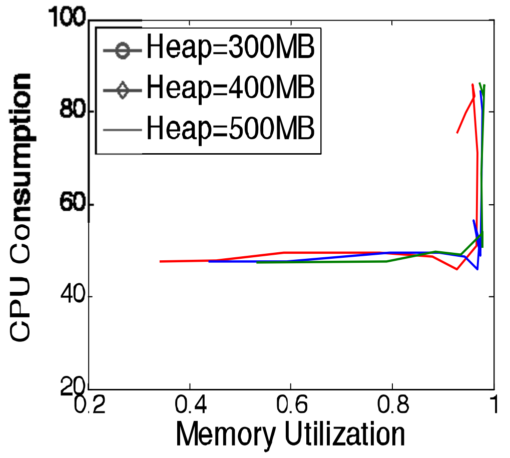
(a)



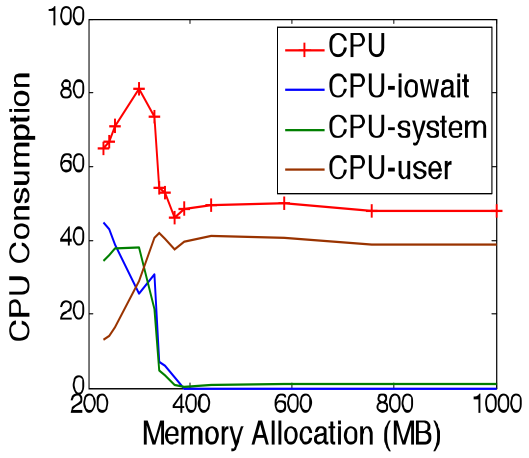
(b)

Figure 3.3: Mean response time (a) and rate of major page faults (b) as a function of memory utilization

sudden surge in the page fault rate when the memory utilization is above 90% confirms our explanation for the relationship we see in Figure 3.3(a). This is an indication of bad interactions between memory allocation and disk I/O that anomalously affect application performance. Such interactions should be avoided or reconciled whenever possible to achieve the desired application performance.



(a)



(b)

Figure 3.4: VM CPU consumption vs. memory utilization for three different heap sizes (a) and breakdown of VM CPU consumption for heap size of 300MB (b)

Figure 3.4(a) shows the average CPU consumption of the VM as a function of the memory utilization. We can see that the VM's CPU consumption remains below 50% for a memory utilization less than 90%, independent of the heap size configuration. However, as the allocated memory is near saturation, we observe significant CPU overhead where the CPU consumption becomes much higher. To further illustrate this, we also show a breakdown of the VM's CPU consumption in Figure 3.4(a), as a function of the memory allocation, for a heap size of 300 MB. As the memory allocation becomes smaller, we first see the total CPU consumption goes up, mainly due to the extra paging activities by the guest OS, as shown by the CPU-system line in Figure 3.4(b). However, if the memory allocation is further reduced, the VM's CPU usage starts to decrease, mainly because the CPU spends a higher portion of time waiting for IO, as the CPU-iowait line indicates. This is also an indication of bad interactions between CPU and memory allocations that should be avoided to achieve good application performance.

In order to understand whether the dynamic CPU control techniques developed in [81, 85] are applicable to controlling memory, we needed to compare the relationship between application performance and memory utilization to the similar relationship for CPU. To this end, we performed a similar set of experiments where the CPU credit scheduler was used to vary the CPU allocation to the VM while using a constant heap size of 300 MB and a memory allocation of 512 MB. In this case, the memory utilization of the VM remained constant, whereas CPU utilization of the VM varied between roughly 20% and 100%. The same experiment was repeated three times, for a different workload intensity of 20, 30, and 40 requests/s, respectively.

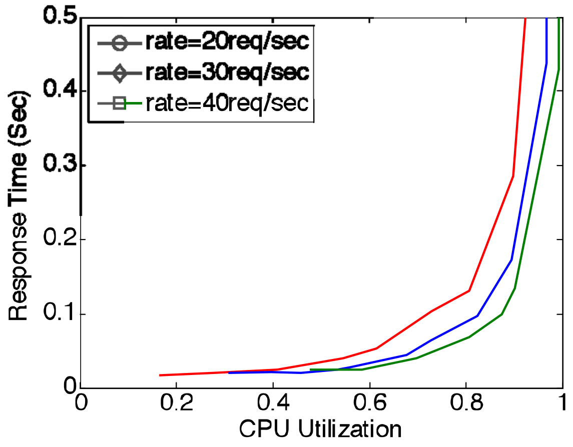


Figure 3.5: Mean response time vs. CPU utilization

Figure 3.5 shows the mean response time as a function of the CPU utilization for the three different workload conditions. By comparing Figure 3.3(a) and Figure 3.5, we can see that these two relationships are similar in that the applications MRT is a monotonically increasing function of either the CPU utilization or the memory utilization. At the same time, we also observe two differences: First, for CPU, the curve that represents the relationship varies with the workload, whereas for memory, the relationship remains almost the same when the applications memory demand changes; Second, the function that represents the relationship is smoother and more differentiable for CPU, but is almost a binary function for memory. This means, when CPU allocation is reduced for an application, there is more graceful degradation in its performance. But we should never allow a VM's memory utilization to go beyond a certain threshold in order to ensure good application performance. When the system falls into the region beyond the threshold, memory allocation may interfere with CPU and disk I/O, which in turn affects response time badly. From Figure 3, we observe that a reasonable threshold to use is 90% for memory utilization.

3.4 Dynamic Memory Allocation

In this section, we present a dynamic memory controller that periodically adjusts the memory allocation to individual VMs such that the application running in each VM can meet its SLO.

3.4.1 Memory Controller Design

Based on the discussion at the end of Section 3.3, it is desirable to maintain the utilization of a VM's allocated memory below a critical threshold. Therefore, we apply the design of the utilization controller previously developed for CPU allocation [81] to dynamic memory allocation. Let $u_{mem}(k)$ and $v_{mem}(k)$ be the memory allocation and usage for a VM during the kth control interval, and let $r_{mem}(k) = v_{mem}(k)/u_{mem}(k)$ be the VM's average memory utilization for the same interval. Then the *Memory utilization controller* uses the following equation to compute the desired memory allocation for the next, or the $(k+1)$th interval:

$$u_{mem}(k+1) = u_{mem}(k) - \lambda_{mem} v_{mem}(k)(r_{mem}^{ref} - r_{mem}(k))/r_{mem}^{ref}. \tag{3.1}$$

This control law has two configurable parameters, where r_{mem}^{ref} is the desired level of memory utilization for the VM, and λ_{mem} is a constant for fine tuning the aggressiveness of the controller. In [81], we have shown that the above controller is stable as long as $\lambda_{mem} \leq 2$. For the experiments in this chapter, we set $r_{mem}^{ref} = 90\%$ and $\lambda_{mem} = 1$. The design of the control law in (3.1) was driven by the bimodal behavior

r_{mem}^{ref}	λ_{mem}	U_{mem}^{min}	U_{mem}^{max}	T_{mem}
0.9	1	230 MB	1024 MB	3 s

Table 3.1: Parameter values of the memory controller

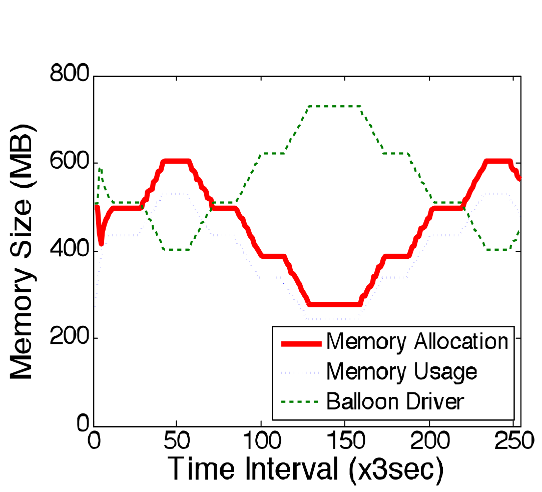
of the system, as shown in Figure 3.3(a). The controller aggressively allocates more memory when the previously allocated memory is close to saturation, and slowly decreases memory allocation in the underload region. This memory utilization controller is fairly easy to implement where the only measurement needed is the memory usage of the VM. The controller also ensures that the value of the memory allocation, $u_{mem}(k+1)$, remains within a specified range of $[U_{mem}^{min}, U_{mem}^{max}]$, where $U_{mem}^{max} = memmax$, the maximum amount of memory the VM is entitled to, and U_{mem}^{min} is the minimum amount of memory to keep the VM running. It is also important to choose a proper control interval, T_{mem}, for the memory controller. We chose the control interval to be as short as possible so that the controller can respond quickly when there is a spike in the VMs memory usage, while taking into consideration the latency in adding and removing memory.

3.4.2 Memory Controller Evaluation

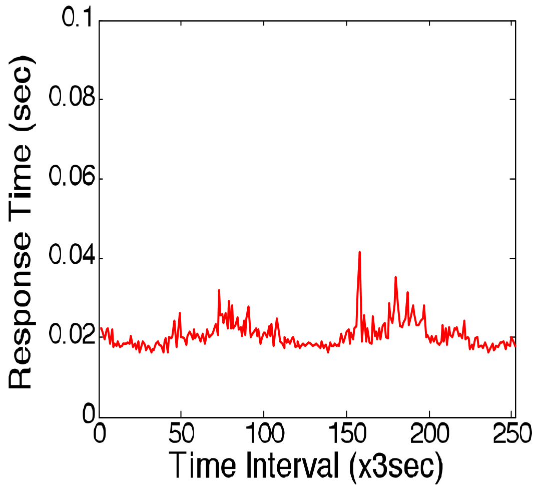
We ran the following experiment to test the performance of the memory controller. We used the *MemAccess* application to emulate a synthetic memory usage trace that has two peaks and one valley and that varies between 256 and 512 MB. Table 3.1 lists the values of the memory controller parameters used in the experiment, and Figure 3.6 shows the results.

The red line in Figure 3.6(a) represents the measured memory usage, and the blue line represents the memory allocation computed by the controller. As we can see, our memory controller does a good job in providing enough memory to the VM, in spite of the varying memory demand from the application. The green line in the figure shows the size of the balloon. This is the amount of memory that can be borrowed by another VM when needed to handle load spikes.

The resulting average response time of the application is shown in Figure 3.6(b). It is evident that the application performance was good throughout the duration of the experiment, where the response time remained below 50ms.



(a)



(b)

Figure 3.6: Memory allocation, usage, and balloon size for the VM (a) and measured response time (b) as a result of the memory controller

3.5 Combined CPU and Memory Control

3.5.1 Nested CPU Controller

We use the nested controllers developed in [85] for CPU allocation to each VM: a *CPU utilization controller* running in the inner loop, which periodically adjusts the CPU allocation to maintain utilization of the allocated capacity at a given target, and a *Response Time controller* in the outer loop, which sets the target utilization in real time based on the observed error between the response time reference and its measurement. Figure 3.7 shows a block diagram of the nested control loops.

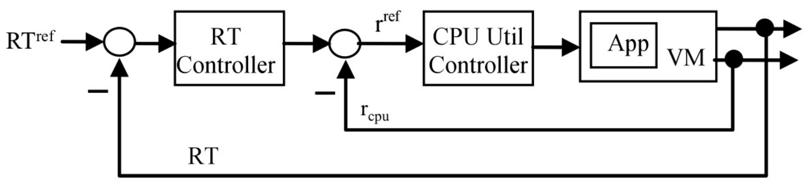


Figure 3.7: Nested CPU utilization controller and response time controller

The CPU utilization of an application is a common metric monitored in production systems to determine whether more or less CPU resource should be allocated to the application. Similar to the notation for memory, let $u_{cpu}(k)$ and $v_{cpu}(k)$ be the CPU allocation and consumption for a VM during the kth control interval, and $r_{cpu}(k) = v_{cpu}(k)/u_{cpu}(k)$ be the VM's average CPU utilization for the same interval. Then the controller uses the following equation to compute the desired CPU allocation for the next, or the $(k+1)$th interval:

$$u_{cpu}(k+1) = u_{cpu}(k) - \lambda_{cpu} v_{cpu}(k)(r_{cpu}^{ref} - r_{cpu}(k))/r_{cpu}^{ref}. \tag{3.2}$$

In this controller, r_{cpu}^{ref} is the desired level of CPU utilization for the VM. The parameter λ_{cpu} is a constant again for fine tuning of the aggressiveness of control actions. The controller is stable for $\lambda_{cpu} \leq 2$ (see [85]). For the experiments in this chapter, we set $\lambda_{cpu} = 1.5$, allowing the CPU controller to be slightly more aggressive than the memory controller. The output of the integral controller is then bounded in the range $[U_{cpu}^{min}, U_{cpu}^{max}]$.

We see from Figure 3.5 that the application's MRT is a monotonically increasing function of the CPU utilization. For a given response time target, RT^{ref}, the corresponding CPU utilization target, r_{cpu}^{ref}, is different for different workloads. For example, the same figure shows that, for $RT^{ref} = 0.1s$, the ideal CPU utilization is 0.73, 0.82, and 0.87 for our workload at 20, 30, and 40 requests/s, respectively. Given that both workload mix and intensities can change at runtime, a second controller is required to translate the value of the RT target to the value of the CPU utilization target automatically in real time. We refer to this controller as the RT controller.

As in [85], we use the following integral control law for the RT Controller:

$$r_{cpu}^{ref}(i+1) = r_{cpu}^{ref}(i) + \beta(RT^{ref} - RT(i))/RT^{ref}, \tag{3.3}$$

where $RT(i)$ is the measured average response time for the ith control interval. The value of the integral gain, β, can be computed based on the approximate slope of the response time curve in Figure 3.5 at the expected operation point, RT^{ref}. For example, if the maximum estimated slope is α, then we need to have $\beta < 1/\alpha$ to ensure stability of the RT controller. The controller also ensures that the r_{cpu}^{ref} value fed to the CPU utilization controller is bounded to an interval $[R_{cpu}^{min}, R_{cpu}^{max}]$.

Note that two different time indices are used in (3.2) and (3.3) to indicate that the utilization controller and the RT controller use different control intervals. In a nested design, typically the inner loop has faster dynamics and therefore uses a shorter control interval than that of the outer loop. Let T_{cpu}^{UC} and T_{cpu}^{RT} be the

λ_{cpu}	U_{cpu}^{min}	U_{cpu}^{max}	T_{cpu}^{UC}	RT^{ref}	β	R_{cpu}^{min}	R_{cpu}^{max}	T_{cpu}^{RT}
1.5	0.1	0.9	3 s	0.1s	0.1	0.5	0.95	12 s

Table 3.2: Parameter values of the CPU controller

control intervals for the CPU Utilization controller and the RT controller, respectively. Then $T_{cpu}^{UC} < T_{cpu}^{RT}$. The difference between the two should depend on how many intervals are required for the inner controller to converge before its reference setting is changed by the outer controller. (See more discussion of this nested design in [85]).

3.5.2 Joint CPU and Memory Controller

As we have seen from many resource utilization traces of real world applications, both the CPU and the memory demands of an application can vary over time. Therefore, our resource controller runs the CPU controller and the memory controller side by side, as shown in Figure 3.1, to ensure that each application running in a VM can obtain enough of both CPU and memory resources such that its SLO can be met.

Note that, unlike the CPU controller, the memory controller is not driven by the response time target in real time due to the sharply different behavior of memory as shown in Figure 3.3(a). Instead, we use 90% as the target for the memory utilization controller to ensure that there is always some amount of free memory available within each VM to avoid memory-related performance degradation in the hosted applications.

Both the CPU and the memory controllers include an arbiter to handle overload situations, where the total computed allocation for the next interval exceeds the resource capacity. In this case, the arbiter reduces the CPU or memory allocation to each VM in proportion to the requested allocation.

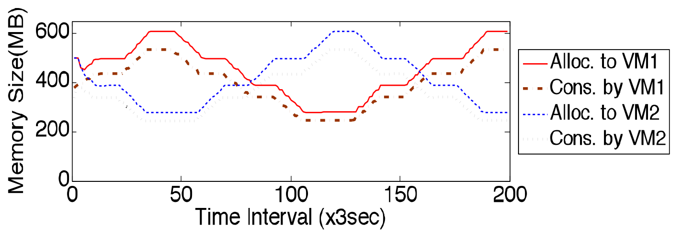
3.5.3 Performance Evaluation Results

We have run performance evaluation experiments of the joint CPU and memory controller on our testbed, described in Section 3.2. In this subsection, we present the results of two such experiments, one using synthetic workload traces, and the other using utilization traces collected from real systems. In these experiments, the parameter values for the memory controller are the same as shown in Table 3.1, and the parameter values for the nested CPU controller are displayed in Table 3.2.

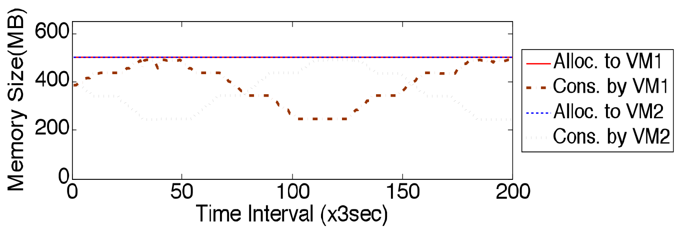
1) Results from Using Synthetic Workloads:

In the first experiment, we ran two Xen virtual machines, VM1 and VM2, on the same physical node, each hosting a MemAccess application. The two guest VMs were pinned to one of the two processors while Dom-0 was pinned to the other processor. In addition, 3GB out of the 4 GB of memory was allocated to Dom-0, and the remaining 1 GB was shared by VM1 and VM2. For each application, we varied both the CPU and the memory loads according to synthetically generated time-varying patterns. For either CPU or memory, the patterns for the two applications are complementary to each other such that the sum of the two peaks is above the capacity of that resource, but the sum of the two instantaneous values is much lower. These workload patterns and resource configurations were specifically chosen to test the performance of our joint CPU and memory controller when both resources are overbooked.

We consider two controller configurations in our experiment. One is the joint CPU and memory controller described earlier, and the other is the CPU controller only without the memory controller. In the latter case, the memory allocation for each VM is statically configured at 512 MB. For both cases, the response time target, RT^{ref}, was set at 100 ms (or 0.1 s) for both applications. The experiment was run for 10 minutes for each controller configuration. The results are shown in Figures 3.8-3.10.



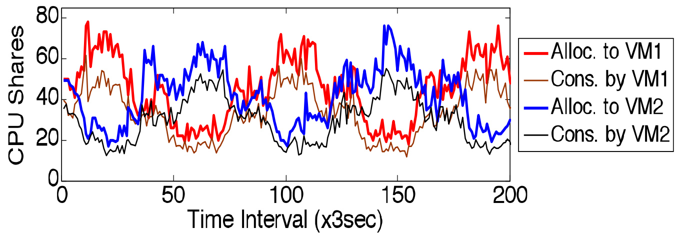
(a) Joint CPU and memory control



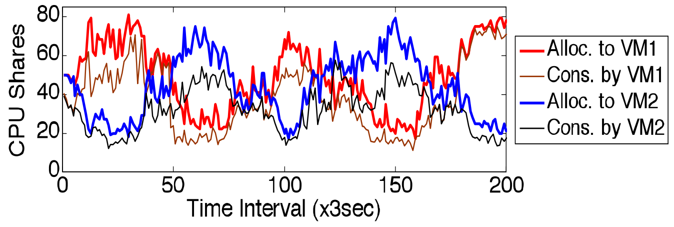
(b) CPU control only

Figure 3.8: Memory allocation and usage for VM1 and VM2 under two controller configurations

Figure 3.8 shows the memory allocation and measured memory usage for both VM1 and VM2 under the two controller configurations. We can see that the memory allocations for the two VMs perfectly track the time-varying memory demands of both applications, when the memory controller was used, as shown in Figure 3.8(a). In contrast, Figure 3.8(b) shows that when no dynamic memory allocation was used, both VMs had their peaks capped by the fixed allocation.



(a) Joint CPU and memory control



(b) CPU control only

Figure 3.9: CPU allocation and consumption for VM1 and VM2 under two controller configurations

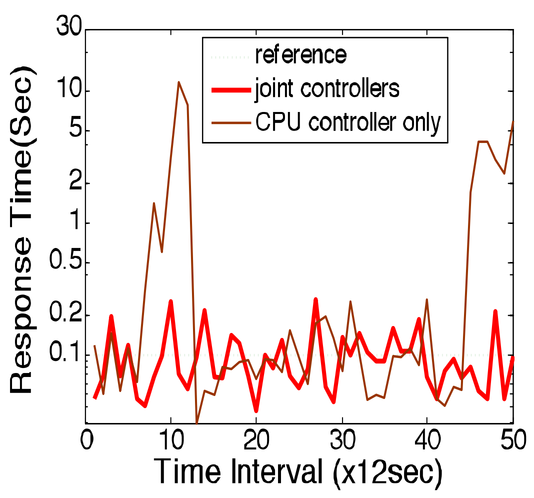
Figure 3.9(a) shows the CPU allocations and consumptions for the two VMs with the joint CPU and memory controller, where the allocations track the consumptions well. Figure 3.9(b) shows the same metrics when only CPU controller was used. We observe that the peak CPU consumption by VM1 was increased from 50% to 70% around the 40th and the 180th time intervals as a result of the memory shortage in this VM.

This is similar to what we observed in Figure 3.4. Consequently, the required CPU allocation to VM1 was also increased, causing the shared processor to be overloaded. The arbiter in the CPU controller was applied to reduce CPU allocations to both VMs.

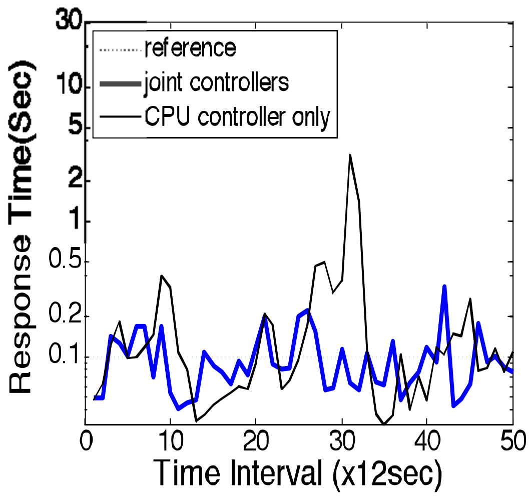
The resulting mean response times of the two applications running in VM1 and VM2 are shown in Figure 3.10. When the joint CPU and memory controller was used, the MRTs of both applications were maintained around their target value of 100ms. When no memory controller was used, both applications experienced significant service level violations where the peak response times were up to two orders of magnitude higher, due to both memory shortage and CPU overload conditions.

2) Results from Using Real World Traces:

In the second experiment, we emulate a scenario where four virtual desktops are consolidated onto a



(a) VM1



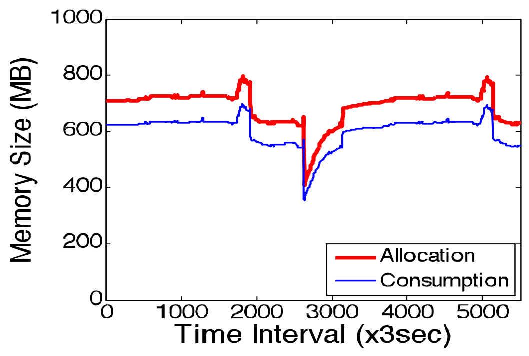
(b) VM2

Figure 3.10: Mean response times (in log10 scale) under two controller configurations for the two applications running in VM1 and VM2

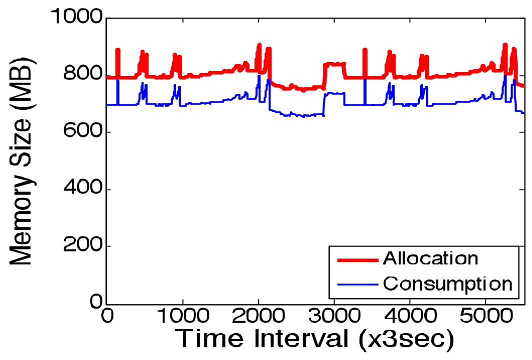
single physical node. To this end, we ran four *MemAccess* applications in four different Xen VMs on the same node. The workloads used for these applications were based on real CPU and memory demand traces collected on four different desktop machines at HP Labs. Each trace contained average CPU and memory consumption for every 1 minute interval during the 9am-2pm window on the same day. This window was chosen such that all the machines were reasonably active where both CPU and memory consumptions demonstrate dynamic behavior. We consider this a more challenging scenario for our resource controller compared to other time periods where either the CPU load was mostly below 10% or the memory usage was flat.

As described in Section 3.2.1, we reproduced both the CPU and the memory demand patterns in each of the applications by dynamically varying both the mean request rate in the workload as well as the heap size allocation to the application once every minute. We also did a capacity planning exercise in advance where we estimated the peak of the total CPU load to be around 85%, and the peak of the total memory load to be around 3.1 GB. Therefore, we allocated one processor to Dom-0 and VM1, and the other processor to the remaining VMs (VM2, VM3, and VM4). We also allocated 512 MB of memory to Dom-0, and the remaining 3.5 GB of memory to be shared by the four guest VMs. The experiment was run for 5 hours, and the results are shown in Figures 3.11-3.13.

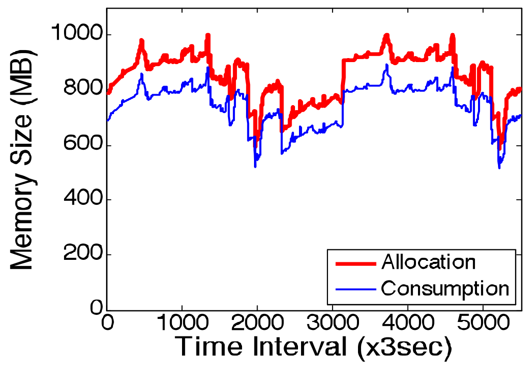
Figure 3.11 shows the memory allocation and usage for the four VMs, and Figure 3.12 shows the CPU



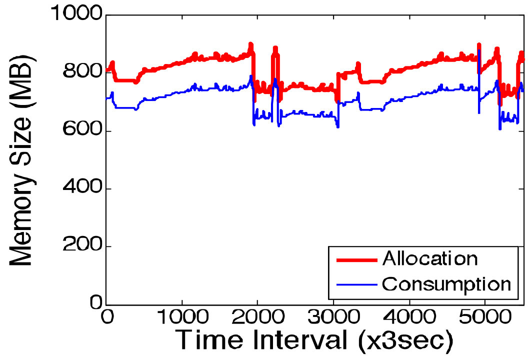
(a) VM1



(b) VM2

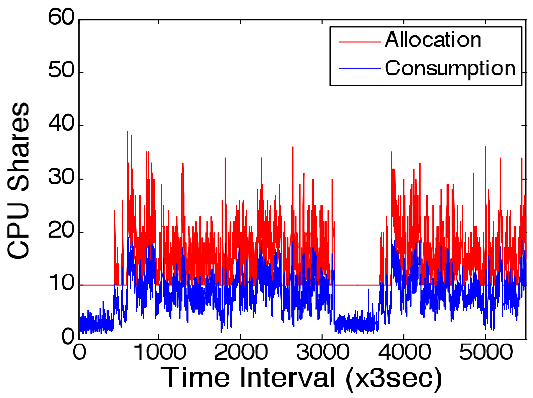


(c) VM3

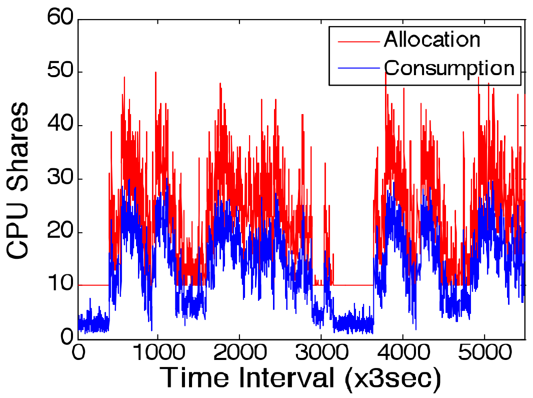


(d) VM4

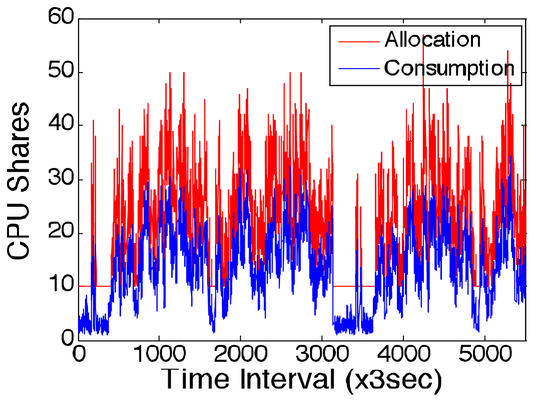
Figure 3.11: Memory allocation and usage for four VMs



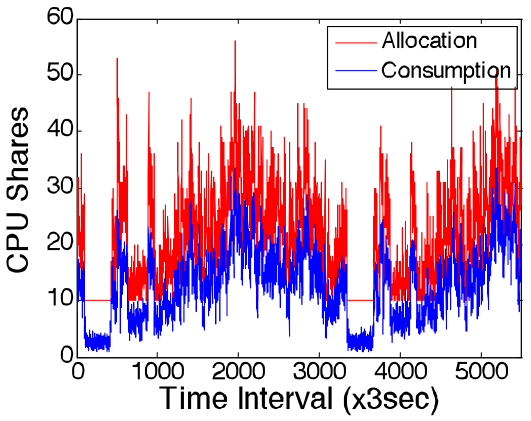
(a) VM1



(b) VM2



(c) VM3

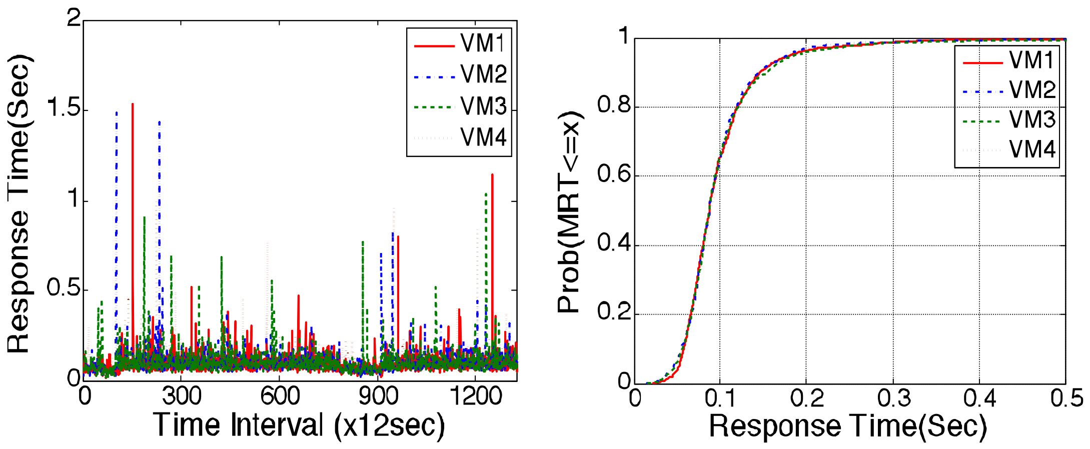


(d) VM4

Figure 3.12: CPU allocation and consumption for four VMs

allocation and measured consumption for the four VMs. Both sets of metrics were collected over the 3-second control intervals.

The measured mean response time for every 12-second interval is shown as a function of the interval number in Figure 3.13(a) for the four applications running in the four VMs. To better assess the overall performance, we also show a cumulative distribution function (CDF) of the MRT for the four VMs in Figure 3.13(b). Interestingly, the CDFs for the four applications are almost identical, although their CPU and memory demands are different. This is likely due to the large number of samples in the data. All the four applications were able to achieve an MRT of 100 ms 65% of the time, and an MRT of 300 ms more than 98% of the time. Such performance is fairly good considering that both CPU and memory are shared among the four VMs, and overall resource utilization of this system is much higher than that of typical enterprise servers.



(a) Time series (b) CDF

Figure 3.13: Measured MRT of the four applications

3.6 Conclusions

In this chapter, we described how OptiTuner can be used to develop dynamic memory allocation to virtual machines on Xen-based platforms. Based on experimental results that characterize the quantitative relationship between memory allocation to a VM and performance of the application running inside the VM, we have designed a joint resource controller combining a utilization-based memory controller and a performance-driven CPU controller by avoiding the region where memory allocation interacts with other components in a way that badly affects application performance. To effectively monitor application performance and control the resources allocated to VMs, the designed joint control system is implemented using OptiTuner that allows an easy access to various sensors and actuators exported by the virtualization layer and the application. Experimental results from the testbed validate that memory overbooking can be achieved using our memory controller, and our joint controller can ensure that all of the consolidated applications can have access to sufficient CPU and memory resources to achieve desirable performance.

Chapter 4

Holistic Approaches for Energy Minimization

4.1 Introduction

In the previous chapter, we presented a case study where OptiTuner was used to implement a joint CPU and memory controller in a consolidated environment. In the case study, it was shown that the interaction between CPU and memory controllers in terms of application-level performance was minimal as long as memory utilization is below a certain threshold. Therefore, the design goal of the joint controller was to stay away from the region in which the interaction between the two controllers arises. However, in many cases, interactions between composed adaptive components are inherent and may not be completely avoided. The best solution in such cases is to proactively coordinate their combined behavior to reduce anomalous interactions, hence achieving the desired performance goals.

Differently from the case study in the previous chapter, the energy minimization application in a Web server farm introduced in Chapter 1 and 2 uses three types of energy saving knobs (DVS, On/Off, BackupAC) that incur constant interactions between them affecting energy consumption of the system. Those knobs should be continuously coordinated in the presence of workload changes to reduce anomalous interactions between them. In this chapter, we implement and evaluate three different holistic approaches for performance management that have been widely used in achieving desired performance in computing systems - centralized optimization [30, 50, 63], distributed optimization [10, 12, 15, 38, 52], and multi-input and multi-output (MIMO) feedback control [17, 78, 79] - in the context of energy minimization in a Web server farm testbed. We show how OptiTuner can be used to apply the three techniques to successfully coordinate control knob adjustment and configure power management for the application.

Our testbed consists of 18 machines with an Amazon-like online book store installed. The performance objective of the resulting application is to minimize the total energy consumption of the server farm, subject to end-to-end delay constraints on client requests and resource constraints on available computational capacity. The testbed is equipped with three different types of performance control knobs to save energy. Through a thorough evaluation process using an industry standard eBusiness benchmark, TPC-W, we show that the three energy minimization approaches can save more energy compared to the baselines, demonstrating that OptiTiuner is indeed useful for developing performance optimization and control for performance-sensitive systems.

The rest of the chapter is organized as follows. We develop an energy minimization application in a 3-tier Web server farm using three different performance management techniques with OptiTuner in Section 4.2. We evaluate OptiTuner in Section 4.3. We conclude the chapter with Section 4.4.

4.2 Three Holistic Performance Management Approaches

In the energy cost minimization application for 3-tier Web server farms, the total cost incurred in the application includes two parts: (i) the energy cost of server machines in the system less (ii) the utility achieved from database backup requests. The goal is to minimize the total cost, subject to the delay constraints of client requests and a resource constraint on the number of available servers. Three types of performance control knobs are used to save energy: On/Off, DVS, and BackupAC. Since their interactions cannot be removed, the three types of energy saving knobs need to be continuously coordinated in the presence of workload changes in a way that reduces bad interactions among them, thereby effectively minimizing energy.

To show the use and efficacy of OptiTuner, we implement three different holistic performance management approaches for the server farm energy minimization application: (i) a centralized optimization approach, (ii) a distributed optimization approach, and (iii) finally a MIMO feedback control approach. We then evaluate and compare the three approaches in Section 4.3.

4.2.1 Centralized Performance Optimization

In this section, we implement a centralized optimization approach for an energy minimization application in server farms using OptiTuner. A centralized optimization approach is a natural choice for achieving good performance given multiple control knobs to tune. The goal of optimization is to find a best combination of the three types of control knobs in the server farm that minimizes power consumption.

Optimization Formulation

We assume that the load is evenly distributed across the machines at each tier in steady state. That is, all m_i machines belonging to a tier i operate at the same frequency f_i. Let $P_{tier}(i)$ be the power consumption of tier i in the system, which is a function of the DVS frequency level and number of machines operating in that tier. λ_{backup} is the rate at which database backup requests are admitted (in cycles/sec). The end-to-end delay bound of client requests is denoted by K. The total number of machines available is denoted by M, and each machine is either turned off or is operating at one of the 3 tiers. It is desired to minimize:

$$\min \quad \sum_{i=1}^{3} P_{tier}(i) - h \cdot \log(\lambda_{backup}) \tag{4.1}$$

$$\text{subject to} \quad \sum_{i=1}^{3} Delay(i) \leq K, \quad \sum_{i=1}^{3} m_i \leq M. \tag{4.2}$$

Observe that the formulation assumes that utility for backup requests increases logarithmically with backup request rate. Power consumption at tier i, $P_{tier}(i)$, is estimated based on the current frequency, f_i, and the number of machines operating at tier i:

$$P_{tier}(i) = m_i \cdot P_{machine} = m_i \cdot (A_i \cdot f_i^n + B_i), \tag{4.3}$$

where B_i is the power consumption that is independent of the current frequency and A_i is a positive constant that indicates the effect of the frequency on power consumption. A_i, B_i, and n can be measured using offline experiments. In general, power changes linearly with frequency and quadratically with voltage [20]. Further, a change in voltage involves a proportional change in frequency. Hence, we will use the value 3 for n for the preliminary results in the next section.

In our testbed, f_i has 8 different discrete values, while optimization works more efficiently when all variables are continuous. We effectively approximate f_i as a continuous variable by replacing it with U_i which is continuous. After getting U_i, we can closely approximate the given U_i by choosing an effective frequency that makes the utilization closest to U_i. This effective frequency can be implemented by time-multiplexing two of the discrete frequencies available on the processor in question. Techniques for doing so have been proposed in earlier literature [51]. To replace f_i with U_i, the steady-state result of an M/M/1 queuing model [47], $utilization = \lambda/\mu$, is used, where μ is the service rate and λ is the arrival rate. We use a queuing model, because it has been widely used for estimating various performance metrics for computing systems such as delay and resource utilization [45]. It yields:

$$U_i = \frac{\lambda_i/m_i}{f_i} = \frac{\lambda_i}{f_i m_i}, \tag{4.4}$$

where λ_i denotes the rate of arrival of requests to tier i, and U_i denotes the utilization of a machine at tier i. Using the relationship shown in Equation (4.4), we rewrite Equation (4.3):

$$P_{tier}(i) = m_i \cdot P_{machine} = m_i \cdot \left(\frac{A_i \lambda_i^3}{U_i^3 m_i^3} + B_i \right). \tag{4.5}$$

Also, λ_{backup} is estimated using the relationship between U_{backup}, m_3, and f_3 from Equation (4.4):

$$\lambda_{backup} = U_{backup} \cdot m_3 f_3 = (U_3 - U_{user}) m_3 f_3, \tag{4.6}$$

where U_{backup} is the utilization of backup requests alone at the third tier, and U_{user} denotes the utilization of user requests.

Assuming the load is equally distributed over the machines at each tier, the delay experienced at each tier i, $Delay(i)$, is estimated using the steady state result of M/M/1 queuing:

$$Delay(i) = \frac{C_i}{f_i - \frac{\lambda_i}{m_i}} = \frac{m_i C_i}{\lambda_i} \cdot \frac{\frac{\lambda_i}{m_i f_i}}{1 - \frac{\lambda_i}{m_i f_i}} = \frac{m_i C_i}{\lambda_i} \cdot \frac{U_i}{1 - U_i}, \tag{4.7}$$

where C_i is a constant in cycles per request to normalize the delay, since f_i is measured in cycles and λ_i is estimated in cycles/sec rather than in requests/sec.

Finally, the resulting cost minimization problem is formulated with the control knobs

$\mathbf{x} = [U_1\ U_2\ U_3\ U_{backup}\ m_1\ m_2]^{\mathrm{T}}$ and the two constraints as follows:

$$\begin{aligned} \min \quad & \sum_{i=1}^{3} m_i \left(\frac{A_i \lambda_i^3}{U_i^3 m_i^3} + B_i \right) - h \cdot \log(U_{backup} \cdot m_3 f_3) \\ \text{subject to} \quad & \begin{array}{c} \displaystyle\sum_{i=1}^{3} \frac{m_i C_i}{\lambda_i} \cdot \frac{U_i}{1-U_i} \le K, \\ \displaystyle\sum_{i=1}^{3} m_i \le M \end{array} . \end{aligned} \tag{4.8}$$

Observe that in the current testbed, the On/Off knob in the third tier is inactive due to data migration problems (we will explain about our testbed in detail in Section 4.3). Hence, m_3 is a constant and the goal of optimization is to find the set of six knob settings, $\mathbf{x} = [U_1\ U_2\ U_3\ U_{backup}\ m_1\ m_2]^{\mathrm{T}}$, that minimizes the cost function subject to the two constraints.

Implementation of Regulation Policies in OptiTuner

We use Shor's r-algorithm to solve the constrained minimization problem shown in Eq (4.8), because the success of the algorithm for nonlinear optimization has been reported recently [8] and its implementation is available in an open source package, OpenOpt [82].

Since it is a centralized solution, we implement a regulation policy running on a dedicated machine that periodically solves the optimization formulation to determine control knob settings at each control period t. One of the machines at each tier is then selected as the leader to run a regulation policy that periodically contacts the central optimization solver (e.g., the regulation policy on the dedicated machine). It sets the control knobs for the next control period $t+1$ for the corresponding tier. The leader is statically defined and specified in the configuration file globally shared by OptiTuner processes.

We also implement tier-wise sensor objects called *TierSensor* that collect performance measurements and resource usage information such as the utilization, request rate, and response time for the corresponding tier that are required to solve the optimization problem. They aggregate performance measurements collected from sensors instrumented in individual machines. Similarly, we implemented two tier-wise actuators, *TierDVSActuator* and *TierOn/OffActuator*, that act on the DVS and On/Off actuators instrumented in individual machines.

When a machine is being turned on or off, there may be some period that it is still consuming power but it cannot serve any new requests. While this overhead can be captured by the problem formulation shown in Eq. (4.8), the worst-case overhead (i.e., toggling a machine every period) depends only on the rate at which the optimization solution is re-computed. Given a fixed rate, the overhead is constant and hence does not affect the solution to the problem. We do not address the rate of recomputing the solution itself as one of the parameters to optimize.

4.2.2 Distributed Performance Optimization

A disadvantage of the above approach is that is is centralized. In this section, we describe distributed performance optimization using OptiTuner. This approach uses optimization decomposition [12], recently applied at length in network theory, to break complex system-wide global optimization problems into less coupled subproblems that can be optimally solved at run-time in a distributed fashion.

Background

Optimization decomposition techniques work best when the constrained optimization problem is convex, in which case a unique optimal solution exists and convergence to the optimal point is guaranteed [6]. When an optimization problem is non-convex in nature, approximate convex models can be used in the optimization formulation. Consider a set of (energy or performance) regulation policies, where x_i denotes the performance control knob setting of regulation policy i, where $i = 1, \cdots, n$ (there are n regulation policies, each manipulating one knob). Let $\mathbf{x} = [x_1, \ldots, x_n]^{\mathrm{T}}$ denote a vector of the knob settings that need to be jointly optimized. A constrained utility maximization problem is formulated as follows:

$$\begin{aligned} \max_{\mathbf{x} \geq 0} \quad & U(\mathbf{x}) \\ \text{subject to} \quad & g_j(\mathbf{x}) \leq 0, \quad j = 1, \ldots, m, \end{aligned} \tag{4.9}$$

where U is the utility function to be maximized; $g_j(\cdot)$, $j = 1, \ldots, m$, are the resource and performance constraints. We assume that the knob settings are real numbers, the utility function $U(\cdot)$ is concave and differentiable, and constraint functions $g_j(\cdot)$ are convex and differentiable.

Introducing Lagrange multipliers $p_1, \ldots, p_m$ for the constraints, the Lagrangian of the problem is given as:

$$L(\mathbf{x}, \mathbf{p}) = U(\mathbf{x}) - \sum_{j=1}^{m} p_j g_j(\mathbf{x}), \tag{4.10}$$

where p denotes the vector of Lagrange multipliers, $\mathbf{p} = [p_1, \ldots, p_m]^T$. The Lagrange multiplier, p_j, is considered as a price for constraint j, indicating the availability of the resource corresponding to constraint j.

The original problem shown in Equation (4.9) has the optimal point at $\mathbf{x}^*$, if and only if there exists the optimal point $(\mathbf{x}^*, \mathbf{p}^*)$ that maximizes the Lagrangian in Equation (4.10) with respect to $\mathbf{x}$ and minimizes the Lagrangian with respect to $\mathbf{p}$. To reach the optimal point $(\mathbf{x}^*, \mathbf{p}^*)$, each regulation policy i periodically adjusts the associated control knob setting x_i to maximize $L(\mathbf{x}, \mathbf{p})$ with regard to x_i individually. In the mean time, each resource constraint monitor j periodically changes p_j to minimize $L(\mathbf{x}, \mathbf{p})$ with regard to p_j, achieving decomposition between the various regulation policies. For example, each period t, each regulation policy i that adjusts the performance control knob setting x_i (to maximize $L(\mathbf{x}, \mathbf{p})$), may use the gradient method (a steepest descent algorithm) [6]:

$$x_i(t+1) = \left[x_i(t) + \alpha_{x_i} \left(\frac{\partial U(\mathbf{x})}{\partial x_i} - \sum_{j=1}^{m} p_j \frac{\partial g_j(\mathbf{x})}{\partial x_i} \right) \right]^+ \tag{4.11}$$

where α_{x_i} is a step size that determines the rate at which the error to the optimal point (x^*, p^*) is reduced[1]. The price value p_j for constraint j is updated in a similar way:

$$p_j(t+1) = \left[p_j(t) - \alpha_{p_j}(-g_j(\mathbf{x})) \right]^+ \tag{4.12}$$

The step size constant affects the convergence rate and the stability of the system. It is shown that smaller step size values help improve system stability, while larger step size values help improve convergence rate [26]. Also the update period should not be too large to promptly cope with the workload changes and closely approach optimality [26].

Thus, the decomposition of the optimization problem enables regulation policies to update their performance control knobs asynchronously and independently of each other. The coordination of regulation

[1] $[Y]^+$ means that the resulting value should be greater than or equal to 0. Arithmetically it is the same as $max(0, Y)$

policies toward the optimal point is done by exchanging the price values of the related constraints and the current settings of the dependent control knobs. This information exchange occurs locally among the dependent regulation policies and the related constraint monitors. The price values adjusted by constraint monitors serve to inform the regulation policies of how much further performance can be improved without breaking the constraints.

The above has some resemblance to market-based resource allocation approaches [58, 77] in that resource availability is expressed as prices and resource allocation is carried out in a distributed fashion based on the prices. However, participants in market-based approaches are usually assumed to be self-interested.

Optimization Formulation and Decomposition into Subproblems

We can use the same optimization formulation shown in Eq (4.8) since it is a convex problem. In order to derive subproblems from the optimization formulation, we first get the Lagrangian of the problem (4.8) as:

$$\begin{aligned} L(x, p_1, p_2) = & \sum_{i=1}^{3} m_i \left(\frac{A_i \lambda_i^3}{U_i^3 m_i^3} + B_i \right) - h \cdot \log(U_{backup} \cdot m_3 f_3) \\ & + p_1 \cdot \left(\sum_{i=1}^{3} (\frac{m_i C_i}{\lambda_i} \cdot \frac{U_i}{1 - U_i}) - K \right) + p_2 \cdot \left(\sum_{i=1}^{3} (m_i) - M \right), \end{aligned} \tag{4.13}$$

where $p_1, p_2 \geq 0$ denote the Lagrange multipliers for the two constraints.

By differentiating the Lagrangian with respect to each of the optimization variables, we can decompose the problem into subproblems that adapt their knob setting individually. Six different regulation policies are created to iteratively adjust the 6 knob settings, $x = [U_1 \; U_2 \; U_3 \; U_{backup} \; m_1 \; m_2]^T$ (m_3 is a constant in our testbed). A constraint monitor is created to adapt the two price values, $p1$ and $p2$, for the two constraints.

By differentiating the Lagrangian with respect to U_1 and U_2, the update equations for the DVS policies at tiers 1 and 2 are obtained as:

$$U_i(t+1) = \left[U_i(t) - \alpha_{U_i} \left(- \frac{3 A_i \lambda_i^3}{m_i^2 U_i(t)^4} + \frac{p_1 m_i C_i}{\lambda_i (1 - U_i(t))^2} \right) \right]^+ \tag{4.14}$$

Once U_i is calculated, the corresponding frequency, f_i, is determined using Equation (4.4). Since the frequency value is discrete in a real system, we choose the closest frequency setting to the calculated value, f_i. Alternatively, a time-multiplexed combination of frequencies can be used as described in [51]. The DVS

Policy at tier 3 updates its knob setting U_3 in a similar way:

$$U_3(t+1) = \Big[U_3(t) - \alpha_{U_3} \cdot \Big(-\frac{3A_3\lambda_3^3}{m_3^2 U_3(t)^4} + \frac{p_1 m_3 C_3}{\lambda_3(1-U_3(t))^2} - \frac{h}{U_3(t) - U_{user}}\Big)\Big]^+. \tag{4.15}$$

The DVS policy at tier i determines its control knob setting, U_i (hence f_i) every second, since the frequency can be changed very quickly. For all tiers, we use 0.05 for the step size constants, α_{U_i}. This value is empirically determined. Formal control-theory analysis of stability of convergence of the above convex optimization approach can be found in [12].

Since the control knob settings U_{backup} and U_3 are related to each other, we calculate U_{backup} based on U_3:

$$U_{backup}(t+1) = \frac{\lambda_{backup}}{\lambda_3} U_3(t+1). \tag{4.16}$$

With the calculated U_{backup}, the BackupAC policy adjusts the portion of the incoming database backup requests accordingly. It uses a control theoretic approach to keep the measured utilization for the backup requests around the set point, U_{backup}, by properly adjusting the admission rate for the incoming backup requests.

The On/Off policies at tiers 1 and 2 update their control knob settings m_1 and m_2 at each invocation period as follows:

$$m_i(t+1) = \Big[m_i(t) - \alpha_{m_i} \cdot \Big(-\frac{2A_i\lambda_i^3}{m_i(t)^3 U_i^3} + B_i + \frac{p_1 C_i}{\lambda_i} \cdot \frac{U_i}{1-U_i} + p_2\Big)\Big]^+. \tag{4.17}$$

The update period for the On/Off policies is set to 1 seconds.

The constraint monitor periodically adjusts p_1 and p_2 respectively using the following update rules:

$$\begin{aligned} p_1(t+1) &= \Big[p_1(t) + \alpha_{p_1}\Big(D(t) - K\Big)\Big]^+ \\ p_2(t+1) &= \Big[p_2(t) + \alpha_{p_2}\Big(\sum_{i=1}^{3}(m_i) - M\Big)\Big]^+. \end{aligned} \tag{4.18}$$

where $D(t)$ is the measured end-to-end delay at time t. Both p_1 and p_2 are updated every second. α_{p_1} is set to 0.05 and α_{p_2} is set to 0.5 (which are empirically determined). In order to accurately calculate p_1, we use a measured end-to-end delay instead of an estimated value using the M/M/1 queuing equation

$\sum_{i=1}^{3}(\frac{m_i C_i}{\lambda_i} \cdot \frac{U_i}{1-U_i})$, because end-to-end delay can be easily measured in our testbed. On the contrary, in the centralized approach, the measured delay cannot be directly used for the delay constraint inequality in Eq. (4.8). Replacing the delay estimation equation with the measured delay would make it impossible to express the dynamics between the delay and control knob values in the optimization process. For example, if the measured delay is less than the bound, K, the inequality becomes inactive so that it has no influence over the optimization process. On the other hand, if the measured delay is larger than K, the optimization solver stops immediately because there is no feasible solution.

The direction of changes in the update rules are opposite to the ones explained in Section 4.2.2, since the objective is to minimize total cost rather than to maximize utility.

Observe that the time scales and step sizes are empirically chosen. Instead of analytically proving the stability of this approach with the time scales and step sizes chosen, we limit the amount of changes in On/Off and DVS knobs to small steps at one period. We observed in the evaluation results (described later in Section 4.3) that the system responds to the approach well, while not losing stability. Formal stability analysis of convex optimization solvers can be found in [12].

Implementation of Regulation Policies in OptiTuner

The decomposition of the optimization problem results in regulation policies to adjust the six different performance control knobs and a constraint monitor to adjust the two price values for the constraints.

Three *DVSPolicy* objects were implemented to perform the adaptation rules, described in Equation (4.14) for the first and the second tier and Equation (4.15) for the third tier. Similarly, two *On/OffPolicy* objects were implemented to perform the adaptation rules specified in Equation (4.17) for the first and the second tier. The *BackupACPolicy* object executes the update rules in Equation (4.16). As all policies work on tiers instead of individual machines, they are placed in the leader machines. The *ConstraintMonitor* object implements the update rules for the two constraints described in Equation (4.18) and runs on the leader machine of each tier. The constraint monitor needs some information (e.g., the average utilization and the number of machines currently used at each tier) from all three tiers to adjust the price values. While this entails that there is exchange of information between all the tiers, the regulation policies at different tiers are still implemented in a distributed fashion and their control knobs are adjusted independently of each other.

As in the centralized optimization approach, the tier-wise sensor *TierSensor* objects, are used to collect performance data and two actuators, *TierDVSActuator* and *TierOn/OffActuator*, are used to enforce the control knob settings.

4.2.3 MIMO Feedback Control

In this section, we implement a MIMO controller using OptiTuner. MIMO control approaches have been successfully used to maintain the desired performance in performance-sensitive systems [17, 78, 79]. The implemented MIMO controller dynamically finds the control knob settings, $\mathbf{x} = [U_1\, U_2\, U_3\, U_{backup}\, m_1\, m_2]^{\mathrm{T}}$ to keep the end-to-end delay constraint. At the same time, by adjusting the three types of power saving control knobs, power consumption is reduced. The MIMO controller designed in this chapter is based on the one used in [78]. We chose it because the MIMO controller they used is directly applicable to our energy minimization application. This also proves the usefulness of OptiTuner because it can be used to implement an existing design technique without difficulties.

Controller Design

In this section, we formally explain the design of the controller. Since the goal of the controller is to minimize the control error, we first define the control error $e(t)$ at each control period t as the difference between the the current average delay $D(t)$ and the end-to-end delay constraint K, hence $e(t) = D(t) - K$. The controller tries to keep the error as small as possible by adjusting the control input (i.e. the control knobs, $\mathbf{x}$).

Before adjusting the control knob values $\mathbf{x}$, they are normalized by subtracting their average, $\mathbf{\Delta x(t)} = \mathbf{x(t)} - \overline{\mathbf{x}}$, where $\overline{\mathbf{x}}$ is the average of $\mathbf{x(t)}$. This is the process of linearlization around the operating point because our controller assumes a linear system. Hence, the designed controller periodically adjusts a control knob change vector $\mathbf{\Delta x(t)} = [\Delta x_1(t), ..., \Delta x_n(t)]$ with given the current error $e(t)$ to minimize the next error $e(t+1)$, where n is the number of control knobs.

In order to effectively design a controller, we need a dynamic model describing the relationship between the control input $\mathbf{\Delta x}(t)$ and the output $e(t)$. We apply *system identification* to derive a linear model for designing our controller as follows:

$$e(t) = \mathbf{A}\, e(t-1) + \mathbf{B}\, \mathbf{\Delta x(t-1)} \tag{4.19}$$

where $\mathbf{A}$ is a $n \times 1$ matrix and $\mathbf{B}$ is a $n \times n$ matrix and n is the number of control knobs.

Using the linear model identified, we design our controller using the Linear Quadratic Regulator (LQR) control [27]. The linear quadratic regulator (LQR) is a well-known control technique that provides practical feedback gains for MIMO control problems. We determine the feedback gain matrix $\mathbf{F}$ for the LQR controller using the matlab *dlqry* function:

$$\mathbf{F} = [\mathbf{K_P} \quad \mathbf{K_I}] \tag{4.20}$$

where $\mathbf{K}_P$ and $\mathbf{K}_I$ are constant matrix. Given the gain matrix $\mathbf{F}$, the controller calculates a control knob change vector $\mathbf{\Delta x}$ at each control period t:

$$\mathbf{\Delta x(t)} = -\mathbf{F}\,[e(t-1) \quad v(t)]^{\mathrm{T}} \tag{4.21}$$

where $v(t)$ represents the accumulated error used for applying the integral gain $\mathbf{K}_I$ to reduce residual error. $v(t)$ is defined as follows:

$$v(t) = \lambda\, v(t-1) + e(t-1). \tag{4.22}$$

where λ is a forgetting factor indicating the importance of the past accumulated error. We use 0.8 for λ.

4.3 Evaluation

In this section, we show the evaluation results of OptiTuner with an energy minimization application in our 3-tier Web server farm testbed of 18 machines. We first describe the implementation details of OptiTuner in Section 4.3.1 followed by the setup of the testbed in Section 4.3.2. In Sections 4.3.5 and 4.3.6, we present results from experiments without and with database backup requests, respectively. We carry out experiments without database backup requests for an appropriate comparison with the approach used in our previous work [30] that didn't consider database backup requests.

4.3.1 OptiTuner Implementation

In this section, we briefly explain the details of OptiTuner architecture and its implementation.

An OptiTuner process runs on each machine involved in the target performance-sensitive system, managing objects residing in the machine. This peer-to-peer property is necessary since modern performance-sensitive systems are usually distributed and large-scale. To provide a uniform access to objects across all OptiTuner processes, a global configuration file is shared among these processes. It contains the information of the registered objects: the location of objects and their properties such as their period (if they are periodically used), measurement value types, control knob value types, and price values.

The *node manager* in an OptiTuner process manages node objects that represent the machines controlled by OptiTuner. Each node object maintains a table for the objects residing in the corresponding machine. In OptiTuner, objects performing periodic operations register themselves with the *event manager*. The event manager invokes the registered objects periodically based on their execution period. Observe that most object classes provided by OptiTuner are periodic. For example, regulation policy objects update their corresponding control knobs periodically and sensor objects usually collect data periodically.

The design of OptiTuner does not limit the choice of the implementation language. The current prototype implementation of OptiTuner is written in Python. Choosing a script language did not affect the performance severely, because most of the computation is done at relatively large periods, hence consuming negligible CPU time. Our measurement shows that OptiTuner implementation written in Python consumes less than 3% of CPU utilization.

4.3.2 Testbed Setup

We constructed a testbed for a three-tier Web server farm with a total of 18 machines. In the setup, we used Apache 2.0 Web server for the first tier, Tomcat for the second tier, and MySQL 5.0 for the third tier. Three machines with Intel Pentium IV 3GHZ CPU and 2GB of RAM are used to run a load balancer for the first tier (Apache 2.3 with mod_proxy_balancer), a load balancer for the third tier (a JDBC-based database clustering solution [14]), a backup request generator, and a client request generator respectively.

Out of the remaining 14 machines with Intel Celeron 2.53GHZ CPU and 512MB RAM, tier 1 and 2 were given 5 machines each and tier 3 was given 4 machines. Each machine belongs to a specific tier and the allocation is static. For example, a machine does not serve as a tier 1 machine part of the time and tier 2 machine at another time. This is in part to avoid the complications of moving machines between different tiers. One might need to implement virtual machine migration to do so. Further, the On/Off Policy in the third tier is inactive in the current testbed, since the database clustering solution does not fully support consistent data migration between replicas. Hence, the configuration of the server farm is: $m_1 \leq 5$, $m_2 \leq 5$, and $m_3 = 4$. Observe that these per-tier upper limits for tier 1 and 2 do not affect the optimization result with the one total machine constraint because the upper limits are not violated with the workload used in the evaluation (explained below).

We used the p4-clockmod driver to change the processor frequency within 8 different frequency levels from 317MHZ to 2.53GHZ. Load balancers for the first and third tiers run on machines with Intel Pentium 4 3GHZ CPU and 2GB RAM. Another machine with an Intel Pentium 4 3GHZ CPU and 2GB RAM was used to generate backup requests for the database tier. MySQL Proxy is installed in the same machine to implement an admission control mechanism for the backup requests. All machines were equipped with Redhat Fedora Core 4 Linux and run OptiTuner as described in Section 4.2.2.

We installed a 3-tier Web application on our testbed based on TPC-W [71], a transactional web benchmark specifically designed for evaluating e-commerce systems. We modified a Java implementation of TPC-W from the PHARM group at the University of Wisconsin [57] to make it compatible with the newest version of Tomcat and MySQL installed in our testbed. The database is configured to contain 10,000 items and 288,000 customers.

In the following experiments, 1500 seconds of TPC-W workload were applied for each test run, with a 300-second ramp-up period, a 1000-second measurement interval, and finally a 200-second ramp-down period. We used the shopping mix workload consisting of 80% browsing and 20% ordering, which is considered the primary performance benchmark in TPC-W. We applied different client loads by changing the number of emulated browsers (EBs). The user think-time between consecutive requests from one EB is set to 1.0 sec. The performance objective is to minimize energy cost subject to the end-to-end delay constraint of 0.5 sec (500 msec) and the resource constraint of the given 14 machines. The remaining three machines for load balancing and backup request generation are not considered when calculating power consumptions.

4.3.3 Power Measurement

We estimate the power consumption of the server farm in two ways. The first method we use is to measure power consumption with external power meters. We use an AC power meter, Watts Up? Pro [19], to measure power consumption in the server farm. 14 server machines are divided into two groups, where the power cords from seven machines are plugged into one AC power meter and those of the remaining are plugged into the other power meter. Each power meter is connected through a serial line to a server machine, periodically recording AC power every second.

We also estimate power consumption based on frequency measurements of the processors. This allows us to evaluate our algorithms on system settings where the effect of DVS on power savings are different. Specifically, the current frequency is measured every sampling interval and we then estimate the achieved power consumption based on the measured frequency.

We first average the power consumption when the maximum frequency level is applied against different offered workload and use it as the base power consumption. Let us call this value $\overline{P_{max}}$. We use the relationship between power and frequency from Equation (4.3) and set the A_i- and B_i-values such that $A_1 = A_2 = A_3 = A$ and $B_1 = B_2 = B_3 = B$, since all the server machines are assumed identical across all tiers. B represents the fixed energy consumption regardless of the current frequency setting and A is a coefficient for calculating the effect of frequency changes (hence the core voltage of the processor). We can then set B as a scaling of the maximum power as $B = \delta\overline{P_{max}}$ where $0 < \delta < 1$. In this way, δ characterizes how much the DVS scaling affects the overall power consumption. A larger δ means that the effect of DVS is less and a smaller δ means that energy consumption is more dominated by the CPU frequency. Then we may calculate A as $A = \frac{\overline{P_{max}} - B}{f^n_{max}}$ Finally, the estimated power $\widehat{P_f}$ with frequency f is

$$\widehat{P_f} = Af^n + B = \overline{P_{max}}\left[(1-\delta)\left(\frac{f}{f_{max}}\right)^n + \delta\right]. \tag{4.23}$$

With given $\overline{P_{max}}$ and the measurement of the current frequency, f, we can calculate the estimated power consumption from this equation. We will use this equation in the evaluation section later in this paper.

4.3.4 Turning Machines On and Off

Turning on or off machines dynamically while the system is serving requests is one of the big challenges to deal with when implementing our proposed distributed energy minimization algorithm. One cannot just turn off a machine when requests are still served, especially if it is a checkout transaction. Removing those sessions might lead to a large loss of revenue and it will also annoy users.

To gracefully turn off a first-tier machine, the front-end load balancer disables a route to the machine. Although the route is disabled, it is still possible that the machine maintains connections waiting for responses from the second tier. Hence, the machine should be turned off only after all the remaining connections are drained out. The middleware checks periodically to see if there are any remaining connections. Turning on a first-tier machine is simply done by enabling the route at the load balancer after the machine is turned on. Turning on or off second-tier machines is done in a similar way to first-tier machines. In addition, to prevent the loss of session information for subsequent requests from a user of the same session, we enable a Tomcat clustering scheme to replicate session information over the second tier.

4.3.5 Experiments Without Backup Requests

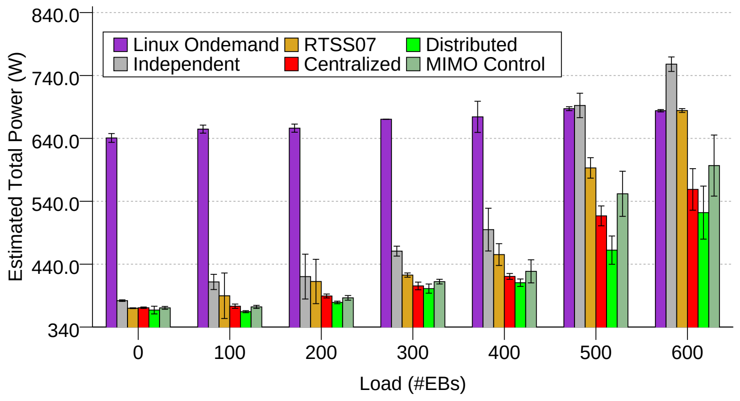


Figure 4.1: Performance (power consumption) comparison between various approaches without backup requests: When B is 70% of the maximum power

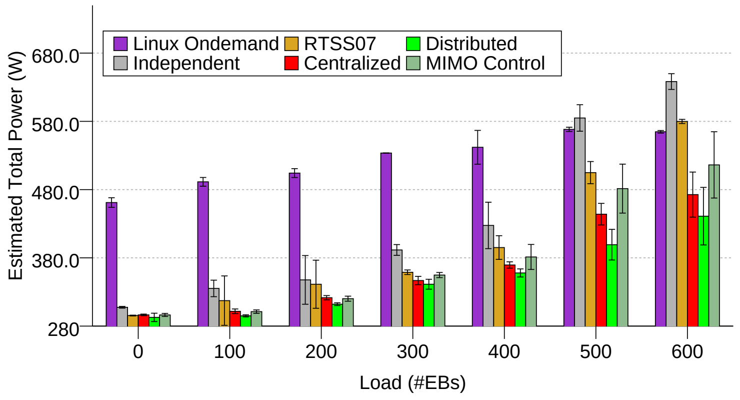


Figure 4.2: Performance (power consumption) comparison between various approaches without backup requests: When B is 50% of the maximum power

In this set of experiments, we consider two different types of performance control knobs, processor frequency level (DVS knob) and the assigned number of machines (On/Off knob) for each tier, hence con-

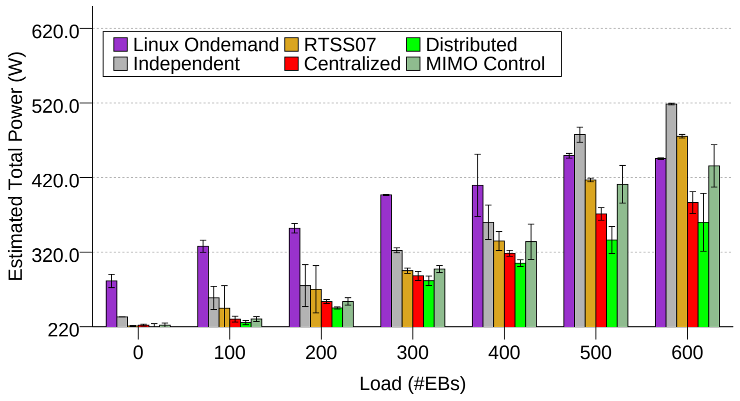


Figure 4.3: Performance (power consumption) comparison between various approaches without backup requests: When B is 30% of the maximum power

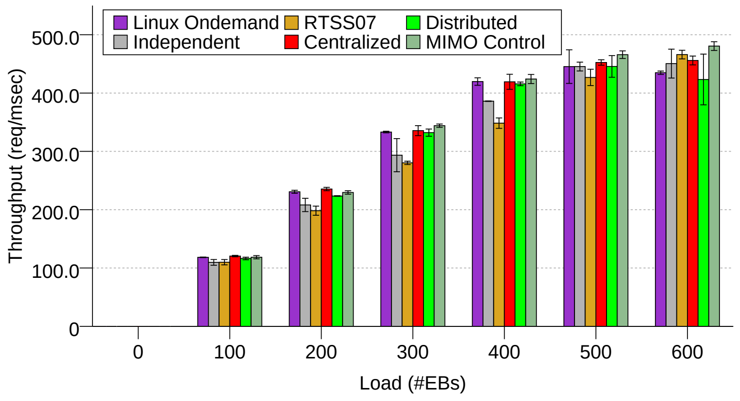


Figure 4.4: Throughput

sidering a total five control knobs (Note that the On/Off knob in the third tier is inactive in the testbed). We evaluate six different energy saving approaches in the absence of backup requests:

- *The Ondemand governor approach* uses only DVS knobs. It uses the Linux Ondemand governor [60], which is a dynamic in-kernel CPU frequency governor that changes CPU frequency levels for energy saving based on CPU utilization.

- *The independent approach* uses independent regulation policies for On/Off and DVS knobs. They determine their knob settings in isolation only on the basis of the current load. Therefore, they are not jointly optimized.

- *The RTSS07 approach*, described in our previous work [30], adapts both types of control knobs. It derives a set of necessary (but not sufficient) conditions for optimality, then uses heuristic-based

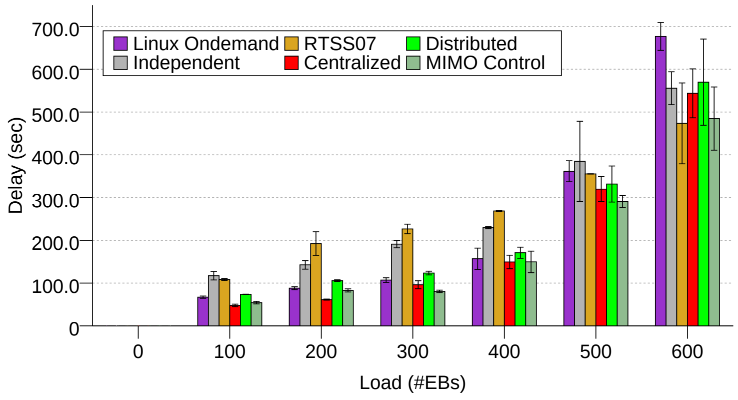


Figure 4.5: Delay

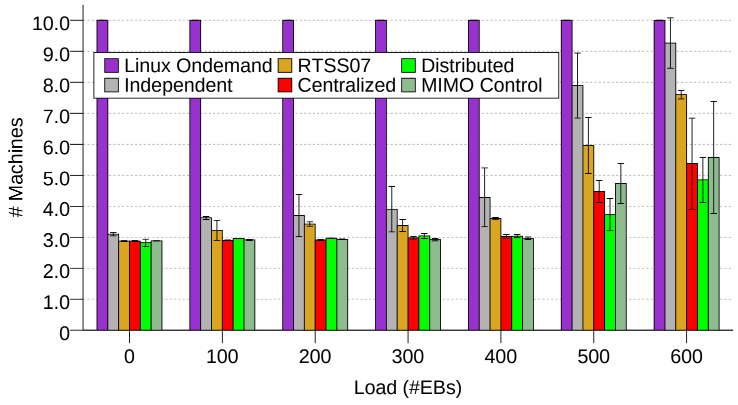


Figure 4.6: Average number of machines for tier 1 and tier 2

feedback.

- *The centralized approach* jointly adapts On/Off and DVS knobs by solving the global optimization problem using the OpenOpt constrained optimization solver (explained in Section 4.2.1).

- *The distributed approach* decomposes the global optimization problem, naturally deriving separate On/Off and DVS policies for On/Off and DVS knobs respectively (explained in Section 4.2.2).

- *The MIMO control approach* adjusts On/Off and DVS knobs determined by the designed MIMO controller (explained in Section 4.2.3).

Among the six approaches, we consider the Linux Ondemand governor approach as the baseline since it is included in the Linux kernel.

Figure 4.1, 4.2 and 4.3, show the power consumption of the various approaches for different values of the

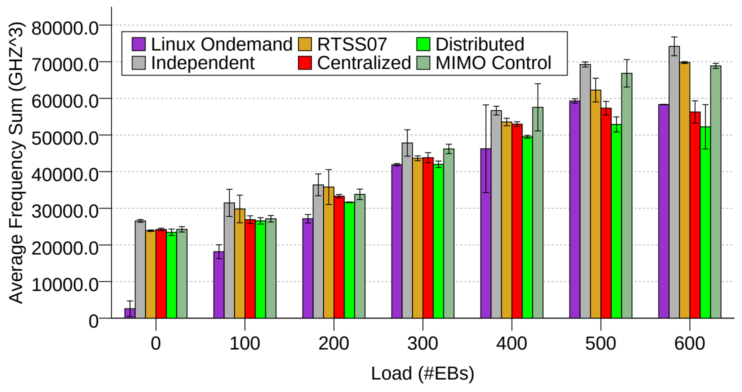


Figure 4.7: Average cube of frequency (f^3) per machine

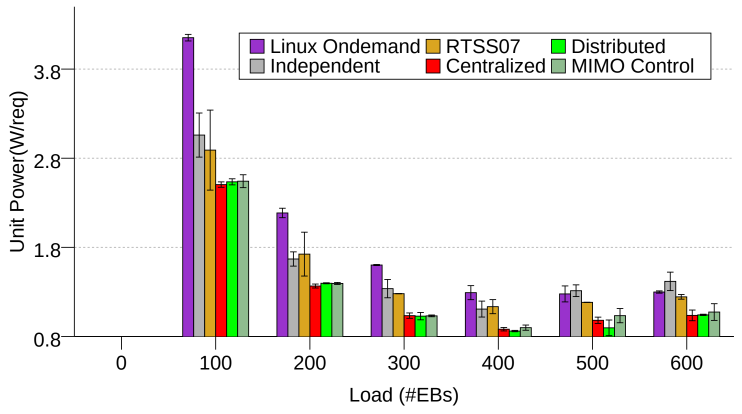


Figure 4.8: Normalized power consumption

parameter B representing systems with different DVS capabilities. The centralized, distributed, and MIMO control approaches save more power in all three different system settings than the other three approaches. For example, at high loads, the distributed approach saves more than 15% energy over the baseline (the Ondemand governor approach). Among the top three approaches, the distributed approach performs slightly better than the centralized and MIMO control approaches.

One of the possible reasons why the distributed approach performs slightly better than the centralized approach is that the distributed approach uses a measured delay when adjusting the price value for the delay constraint. On the contrary, the centralized approach estimates delay using the model shown in Eq. (4.7) when solving the minimization problem. Therefore, the distributed approach can adapt to workload changes more accurately, leading to a better performance. Further, the distributed approach performs a little better than the MIMO control approach because the MIMO control approach presented in this paper does not aim to minimize power. Its goal is to keep the end-to-end delay around the set point, while doing so can still considerably save power consumption.

The RTSS07 approach comes next to the centralized, distributed, and MIMO control approaches because it uses a heuristic feedback approach to search for the optimal point. The baseline, the Linux Ondemand governor approach, has higher energy consumption as it uses only DVS knobs. The independent approach adjusts On/Off and DVS control knobs independently, incurring higher energy consumption at high loads since they are not jointly optimized.

Figure 4.6 shows the average number of machines used during the experiments. We only consider tier 1 and tier 2 machines to calculate averages because the number of machines at tier 3 is fixed. The Linux Ondemand governor approach shows the largest values as expected, since they use a fixed number of machines. Interestingly, at high loads, the independent approach uses almost all available machines, hence spending more power. As introduced in Section 4.1, this is an instance of undesirable interactions between regulation policies when executed independently. This result shows the need for design techniques to coordinate the adjustment of control knobs. The centralized, distributed, and MIMO control approaches show smaller average values than the other three approaches confirming their control knob adjustment is coordinated.

Figure 4.7 shows the average cube of frequency (f^3, where f is CPU frequency) per machine. We use f^3 because the power equation Eq. (4.3) uses it to calculate power consumption. Overall, the independent approach shows the highest frequency sum because of uncoordinated control of control knobs.

Figure 4.4 and 4.5 depict throughput and end-to-end delay respectively. All approaches show similar performance in terms of throughput and end-to-end delay. But the Ondemand governor approach shows a slightly higher delay at high load (e.g., when EB is 600) than the other approaches because it only considers CPU utilization when changing CPU frequency.

Finally, Figure 4.8 depicts the normalized power consumption of the presented approaches calculated as power consumption divided by throughput. The three approaches - the centralized, distributed, and MIMO control approaches - show smaller normalized power consumption than the other three approaches, proving that throughput is not sacrificed to achieve power savings.

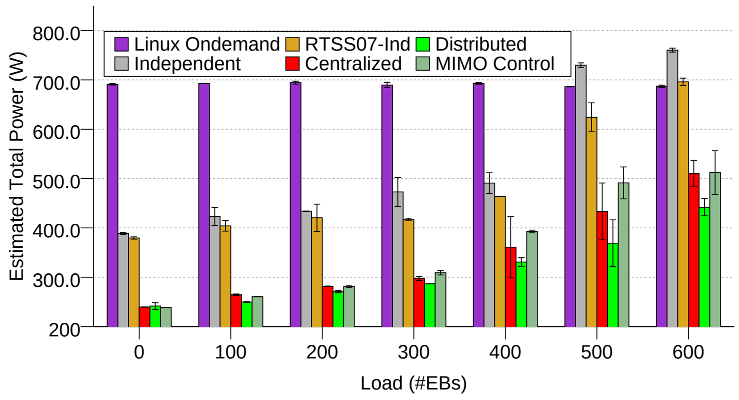


Figure 4.9: Cost comparison between various approaches without backup requests: When B is 70% of the maximum power

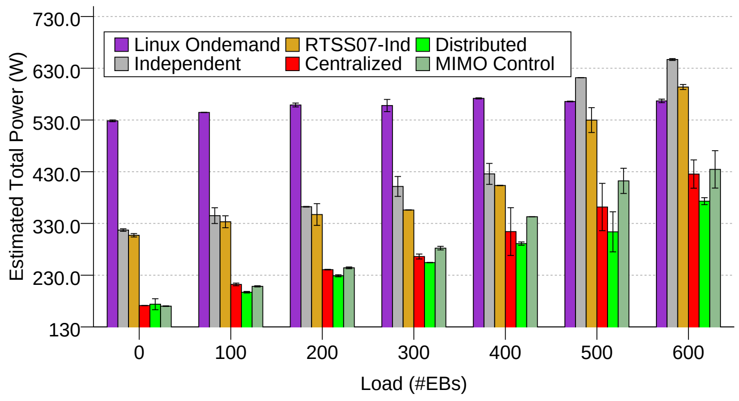


Figure 4.10: Cost comparison between various approaches without backup requests: When B is 50% of the maximum power

4.3.6 Experiments with Backup Requests

In this set of experiments, we consider database backup requests that result in an additional performance control knob, BackupAC knob, that determines the admission rate for the backup requests in addition to DVS and On/Off knobs. We evaluate six different energy saving approaches: the Linux Ondemand governor approach (the baseline), the independent approach, the RTSS approach with an independent BackupAC policy (we call RTSS07-independent), the centralized approach, the distributed approach, and finally, the MIMO control approach.

- The Ondemand governor approach uses only DVS and BackupAC knobs. The Linux Ondemand governor (for DVS knob) and BackupAC policy (for BackupAC knob) independently adjusts their control knobs. The BackupAC policy controls the incoming backup request rate based on the end-to-

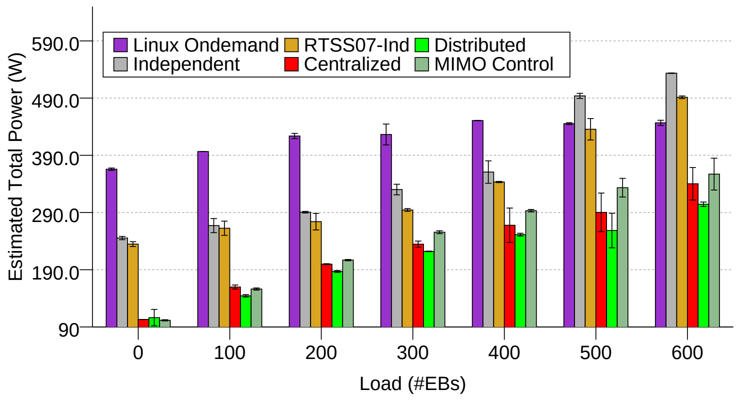


Figure 4.11: Cost comparison between various approaches without backup requests: When B is 30% of the maximum power

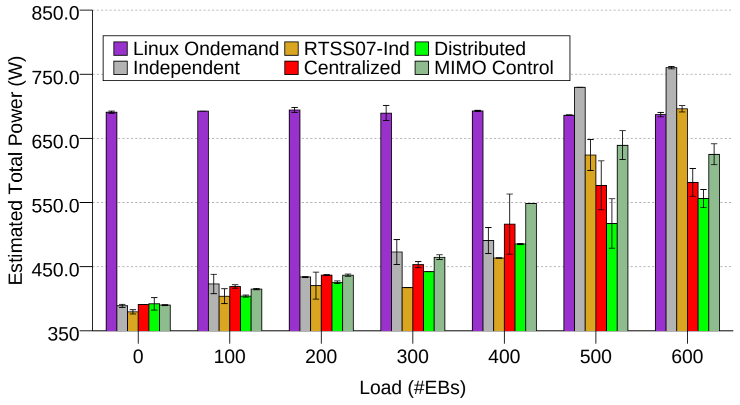


Figure 4.12: Power comparison between various approaches without backup requests: When B is 70% of the maximum power

end delay constraint regardless of the decision of the Ondemand governor.

- The independent approach uses independent regulation policies for On/Off, DVS, and BackupAC knobs. All regulation policies independently determine their knob settings based on the current load.
- The RTSS07-independent approach combines the RTSS07 approach that jointly adapts DVS and On/Off knobs, with a BackupAC policy for database backup requests that runs independently of the other two control knobs.
- The centralized approach jointly adapt DVS, On/Off, and BackupAC knobs by solving the global optimization formulation (explained in Section 4.2.1).

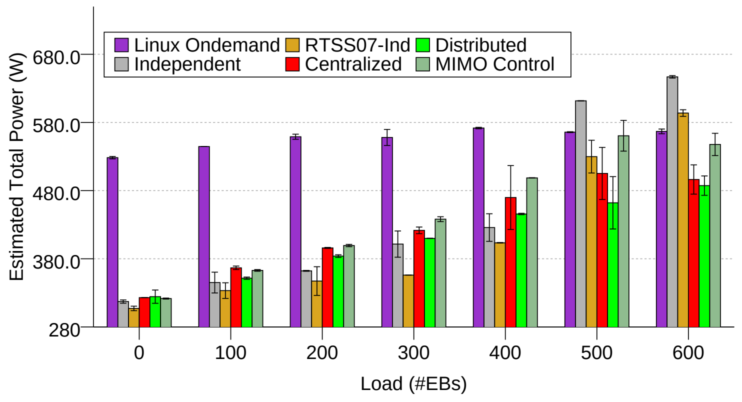


Figure 4.13: Power comparison between various approaches without backup requests: When B is 50% of the maximum power

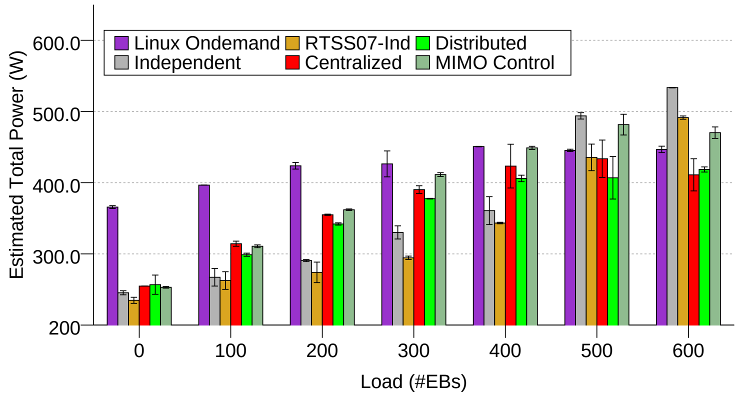


Figure 4.14: Power comparison between various approaches without backup requests: When B is 30% of the maximum power

- The distributed approach has the DVS, On/Off, and BackupAC policies for the control knobs co-adapted as per the optimization decomposition methodology described (explained in Section 4.2.2).
- The MIMO control approach adapts DVS, On/Off and BackupAC knobs as determined by the MIMO controller designed (explained in Section 4.2.3).

For different system settings obtained by varying the parameter B, the overall cost (calculated with Eq. (4.8)) incurred by the three approaches - the centralized, distributed, and MIMO control - is less than that of the other three approaches as shown in Figure 4.9, 4.10, and 4.11, because they serve more backup requests while incurring comparable or less power consumption. When the workload is not high, the power consumption of the centralized, distributed, and MIMO control is comparable to that of the independent and RTSS07-independent approaches as shown Figure 4.12, 4.13, and 4.14. However, the centralized,

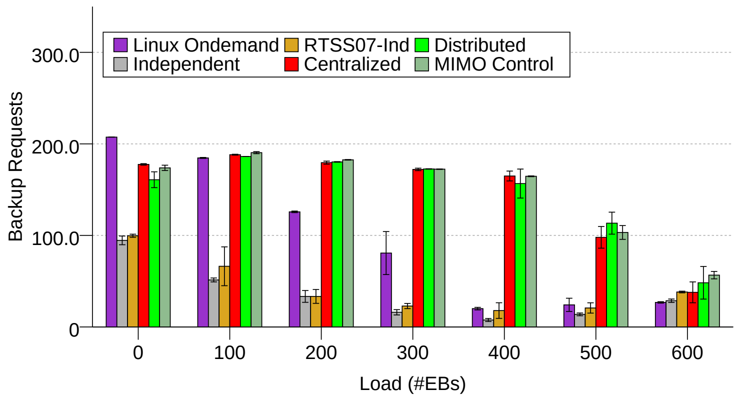


Figure 4.15: Served Backup Requests

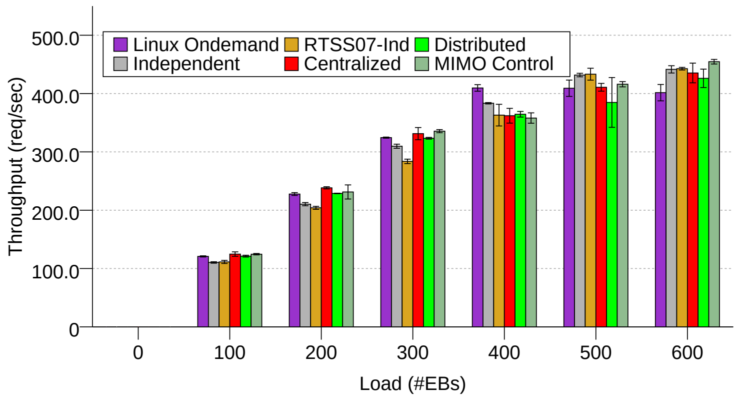


Figure 4.16: Throughput

distributed and MIMO control approaches accept more backup requests as shown in Figure 4.15. This can be attributed to the fact that those three approaches successfully minimize undesired interactions between the three control knobs, saving energy cost which is a function of energy consumption and the number of served backup requests.

While the Linux Ondemand approach performs poorly at low workloads (has more machines turned on than required) in terms of cost, it performs better than the independent and the RTSS07-independent approaches at high load (see Figure 4.9, 4.10, and 4.11). This is because, as the workload increases, the settings of all three control knobs are not jointly adapted in these approaches. Hence, they incur more cost than the Linux Ondemand approach that uses only two control knobs.

This ill-coordinated behavior at high workloads in the independent and the RTSS07-independent approaches, becomes obvious from Figure 4.15 and 4.18. At high loads, the number of backup requests served by these approaches actually increases with system load as shown in Figure 4.15. Further, the two ap-

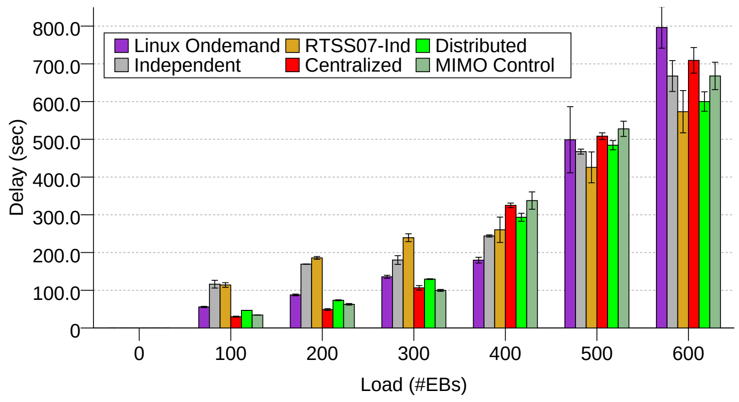


Figure 4.17: Delay

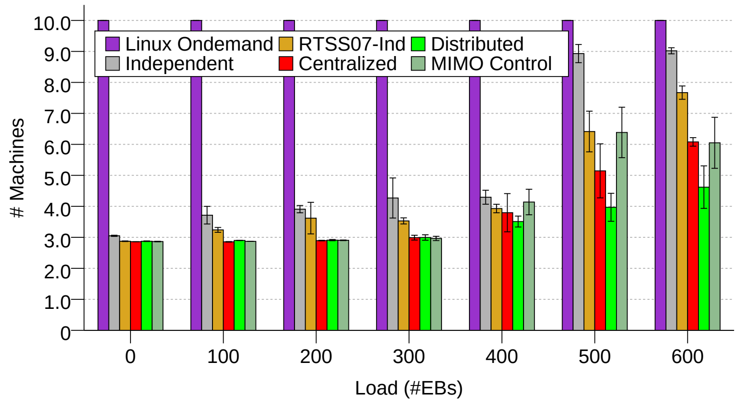


Figure 4.18: Average number of machines for tier 1 and tier 2

proaches use more machines at high loads as shown in Figure 4.18. Especially, the independent approach uses almost all available machines. These are also instances of undesirable interaction between multiple control knobs when they are adjusted independently.

For the same experiment, the throughput and end-to-end delay are shown in Figure 4.16 and 4.17 respectively. The three approaches - the centralized, distributed, and MIMO control - show comparable performance with the other schemes with respect to all these metrics, showing that the three approaches do not compromise on throughput or delay in order to achieve the reduced energy cost. By admitting more backup requests when the user load is low, and by always coordinating adjustment of control knobs, they save considerable amount of energy cost compared to the baseline.

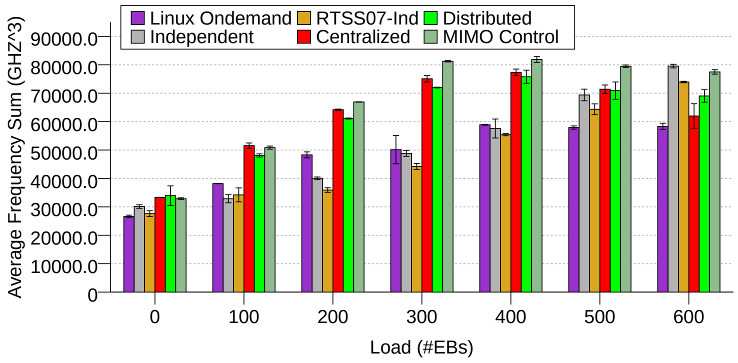


Figure 4.19: Average cube of frequency (f^3) per machine

4.3.7 Discussion

While all three approaches - the centralized optimization, distributed optimization, and MIMO control approaches - successfully achieve good aggregate behavior in terms of adjusting multiple control knobs, there exist differences between them.

While the centralized and distributed approaches try to minimize power consumption, the main goal of the MIMO feedback approach presented in this work is to maintain end-to-end delay around the set point (e.g., the end-to-end delay constraint). Although it doesn't explicitly try to minimize power consumption, in order to keep the end-to-end delay around the set point, the MIMO feedback approach lowers down CPU frequency and turns off unnecessary machines when the system is underutilized, hence saving power consumption.

Further, the distributed approach differs from the other two approaches in that the decisions of the other two approaches are performed in a centralized way while the distributed approach determines control knobs settings in a distributed way.

This distinction may have scalability implications. In principle, in very large farms, it could be that a centralized solution will eventually cause a bottleneck. Hence, the distributed solution may be preferred. At the same time, the performance of the distributed solution may start degrading with scale. Unfortunately, we are not in possession of a data-center scale testbed where we could experiment with life-size systems. The point from the paper is merely to offer OptiTuner as a service that would allow system administrators to easily evaluate the pros and cons of the different approaches on their systems.

4.4 Conclusion

In this chapter, in order to show the efficacy of OptiTuner, we applied it to implement and configure three different holistic performance management approaches that have been widely used in achieving desired performance in computing systems - centralized optimization, distributed optimization, and MIMO feedback control approaches - for an energy minimization application in a Web server farm. With the abstractions provided by OptiTuner that encompass common operations of different performance management techniques, the three holistic approaches were easily implemented and configured to run in OptiTuner. Results from the evaluation in a server farm testbed with 18 machines showed that those approaches were able to reduce total energy consumption considerably compared to the baseline approaches through intelligently coordinating control of multiple performance knobs.

Chapter 5

AdaptGuard: Online Diagnosis and Recovery Service

5.1 Introduction

While design-time and integration-time tools, developed in this dissertation, will significantly reduce the possibility of software instability at run-time, perfection cannot always be assumed. Hence, subtle unstable interactions may occur that result in performance problems. Such interactions must be corrected at run-time. Differently from performance problems caused by component failures or resource bottlenecks, this dissertation focuses primarily on performance problems caused by self-reinforcing interactions between subsystems that lead to bad states [28, 30, 54, 55].

Since individual components are easier to test in isolation, errors that manifest themselves within a single component are usually removed early in the development process. The residual errors not identified at system design and integration time are those that occur because of the way components interact. For example, undocumented assumptions on acceptable inputs might be broken when different modules are combined. The cause of performance problems resulting from such bad interactions often cannot be easily diagnosed by previous debugging approaches geared for detecting single component failures [11, 80], or those geared for isolating performance bottlenecks [2, 13, 43].

Such unexpected bad interactions may occur even in well-managed commercial systems. Recently, Gmail took some of their servers offline to do some maintenance, which made some of the servers overloaded. By not allowing traffic to flow into the overloaded servers, the problem got worsened as the traffic had to be directed to an even smaller number of servers. Eventually all servers stopped accepting requests. Similarly, Facebook was down for a few hours as an automated system for verifying cache values conflicted with an one-time persistent copy changes for error correction. As the persistent copy changed, the automated system treated all changes as errors and thereby overwhelmed the back-end database system. This created "real" errors due to the overloaded condition, aggravating the situation.

To address design of automatic problem diagnosis and recovery, we present *AdaptGuard*, an online diagnosis and recovery service for performance-sensitive systems, that guards the target system from instability without the benefit of a priori system models. The online diagnosis service is implemented by using OptiTuner, thereby running as a separate software service to properly monitor the target performance-sensitive system. It can detect transient errors and performance problems arising from residual unexpected side-effects of module composition and recover from instability caused by such problems when possible.

The rest of the chapter presents two different diagnosis mechanisms that AdaptGuard provides. Section 5.2 presents a mechanism that uses a simple statistical correlation analysis. It *automatically* infers the implicit assumptions a designer must have made that have a bearing on loop stability. When they are violated, it expects instability to occur and performs intervention, breaking the actual runaway loop. This mechanism is, however, constrained to systems where correlation analysis between single events is sufficient for solving the problem. We also consider problems where order of events is important as well. Section 5.3 describes a mechanism that extends the first mechanism to detect discrete chains of events uses data mining techniques and identify anomalous actions causing performance problems. It leverages *discriminative sequence mining* using logs of correct past behavior in order to identify, by contrast, any recent anomalous chains of events that are candidates for blame for the performance problem.

5.2 Online Diagnosis and Recovery Service

Adaptive components in performance-adaptive systems are carefully designed by domain experts to prevent the system from becoming unstable. Naturally, the design of adaptive components makes implicit assumptions about their effect on other system components and the external environment. For example, one may assume that turning off a server may increase load on its mirrors [30]. When such seemingly correct assumptions are violated, the system may respond inappropriately to external stimuli possibly driving performance in the "wrong direction" in ways not predicted by the designer. When the problem occurs, the mission of AdaptGuard is threefold. In the absence of a user-supplied model that describes the system, AdaptGuard should (i) anticipate imminent performance degradation, (ii) attribute it correctly to the responsible chain reaction, and (iii) stop it. It is key to notice here that since performance degradation may occur in ways not predicted by the designer, it might not, in fact, be measurable (e.g., because the designer did not anticipate to measure the right variable). A key requirement of AdaptGuard is therefore to fulfill its threefold mission without the benefit of actually observing the performance degradation. AdaptGuard must use means other than direct observations to anticipate degradation. This distinguishes it from the trivial case where a control-loop-gone-bad is stopped because of a *measured* performance problem.

Using the interfaces provided by OptiTuner, any regulation policy can be connected to any performance sensor or actuator registered by the programmer. Hence, AdaptGuard knows which sensor-regulation policy-actuator loops are in place. It then uses a simple heuristic to monitor stability of such loops. Unstable loops are opened and stability is restored since software systems are generally open-loop stable.

To better illustrate our approach, we implement a QoS-adaptive Web server testbed using AdaptGuard. We implement two typical regulation policies frequently encountered in the literature [30, 36]: an admission control policy and a power saving policy. We then artificially cause assumption violations by injecting software faults. We first consider two simple software faults, the missing file and the busy loop faults, followed by an interesting case where the two regulation policies are combined together creating instability. Our evaluation results demonstrate the efficacy of AdaptGuard in detecting assumption violations caused by the injected faults and restoring acceptable performance.

Further, we present a running case study to demonstrate that AdaptGuard is also useful in real-world scenarios. We present an admission controller for Web servers that preferentially drops lower priority requests over higher priority ones under overload. We show, however, that due to an interesting kernel-level mechanism [65] to protect against livelock and denial of service attacks, the feedback control loop of the admission controller destabilizes, causing the server to plunge deeper into overload and dropping indiscriminately a significant fraction of both low-priority and high-priority requests. When AdaptGuard detects preconditions of instability (namely, positive feedback), it takes action to stop the offending feedback loop. Correspondingly, it is shown that the server is able to provide better service to high priority clients.

The rest of the section is organized as follows. Section 5.2.1 highlights key features of AdaptGuard. Section 5.2.2 describes the design of AdaptGuard. Section 5.2.3 evaluates our approach. Finally, we conclude with Section 5.2.4.

5.2.1 Overview

Performance adaptation modules (i.e. regulation policies) make assumptions about the effect of their corrective actions. When these assumptions are violated due to system anomalies or faults that were not captured in the assumptions, software systems exhibit poor performance. The primary concern of AdaptGuard is to detect such violations and recover acceptable performance of the target system upon assumption violations.

When building performance-adaptive systems with AdaptGuard, the implementation of regulation policies (using sensor, regulation policy, and actuator object interfaces provided by AdaptGuard) can be easily and automatically translated into adaptation graphs that represent the underlying causality assumptions. AdaptGuard does not understand the user implementation of these objects. It merely implements the interface that carries communication between them. This communication happens to convey the performance variables being measured and controlled. It implements a causality chain from measurements to responses. This chain is, in essence, a feedback control loop, closed by a target application (e.g. process) that is monitored by AdaptGuard. Feedback control theory tells us that a stable feedback control loop must be negative. In contrast, positive feedback is unstable.

With the observation, AdaptGuard uses a heuristic-based approach. Given the communication chain between sensors, regulation policies, and actuators that a programmer implements (using the interfaces exported by AdaptGuard), it monitors the correlations between variables in the chain. When any correlation coefficient changes sign (or an odd number of them in the same chain do), the sign of the corresponding adaptation loop must have changed and hence, the loop must have become unstable. AdaptGuard learns the correct signs of correlation coefficients by monitoring system execution for a sufficiently long amount of time. This approach assumes a normally-correct system, which is reasonable because AdaptGuard is geared for protecting deployment-ready systems, as opposed to those in their early stages of debugging. It takes only a few minutes to learn the right coefficient signs, which is negligible compared to the time-to-failure of modern embedded and server systems. Hence, AdaptGuard discovers the causality assumptions made in the design of an regulation policy without needing explicit user input.

5.2.2 Design

Adaptive systems make assumptions regarding the external effects of adaptive actions. We call these assumptions *causality assumptions*. For example, when designing QoS-adaptive Web servers, we know (from queuing theory) that the response time, D, decreases with increased system speed (e.g., CPU service rate), μ, and increases with increased workload (e.g., request arrival rate), λ. This queuing-theoretic fact is a common foundation to much prior work on server admission control [36], performance adaptation [1], and energy control [30].

While queueing theory is correct, a designer might not always take everything into consideration. For example, a designer's model might have ignored a secondary effect (such as virtual memory swapping overhead or a limit on the maximum number of open file descriptors). Ordinarily, the secondary effect does not appreciably affect performance, but it may become dominant in certain corner cases thus invalidating the model that ignored it. Since most models are merely abstractions of the actual implementation, approximations inevitably exist and may cause assumption violations. Assumption violations may cause feedback (i.e., adaptation) loops to become unstable.

Section 5.2.2 defines more formally what we mean by causality assumptions that AdaptGuard automatically identifies and explains the procedures of inferring causality assumptions in Section 5.2.2. Section 5.2.2 reviews the notion of adaptation graphs. Section 5.2.2 describes our mechanism for detecting such violations at runtime. Recovery is described in Section 5.2.2.

Causality Assumptions

A causality assumption, $A \to B$, (i) states that changes in variable A cause subsequent changes in variable B, and (ii) indicates the direction of change (i.e., whether the two variables change in the same direction or in opposite directions). This definition is geared towards computing systems where most relations between parameters are algebraic (as opposed to, for example, relations expressed by differential equations). While estimation and control theory offer much more precise tools for model estimation (such as Kalman filters and recursive least squares estimators), our goal is to allow for a wide range of linear and non-linear functions to be represented by a simple and general model. A violation of assumption, $A \to B$, therefore occurs if and only if either the causal relation between the two variables is broken (e.g., the two variables become independent), or the direction of change is reversed (e.g., they become inversely proportional instead of directly proportional).

Focusing on the stability of adaptation loops (feedback control loops), AdaptGuard only considers the variables used in adaptation loops for inferring causality assumptions. Since AdaptGuard implements the interface between performance sensor, actuator, and regulation policy objects, once the application designer has implemented an adaptation loop, AdaptGuard can identify the variables involved without actually knowing the implementation details (e.g. the designer's model for implementing the loop). It then monitors the identified variables to detect the relation between them. Causality assumptions therefore can be automatically inferred by AdaptGuard from the structure of the system implementation and a small amount of run-time monitoring.

Observe that, in terms of identifying the variables of interest for inferring causality assumptions, AdaptGuard is significantly different from the previous approaches that make substantial efforts to discover variables of interest and their correlations from a large number of variables extracted from system logs and measurements [35, 48]. Those techniques, however, may be used by application designers, for example, to determine the key variables that affect system performance when designing an adaptation loop.

Inferring Causality Assumptions

With the identified variables (that are visible to regulation policies), AdaptGuard infers (pair-wise) causal relations among the variables. Given a pair of variables, x and y, AdaptGuard tries to figure out if causal relations, $x \rightarrow y$ and $y \rightarrow x$, exist. It calculates the correlation coefficient between the variable at the head of the arc and the time-displaced (into the past) variable at the tail, since past values of the latter causally affect the former. The correlation coefficient tells us the relationship between the two variables. A value of 1.0 corresponds to a perfectly positive correlation and -1.0 represent a perfectly negative correlation. Zero (or a very small value) indicates that no (or very small) correlation exists. The correlation coefficient between variables (time-displaced) x and y is estimated at the kth period as follows:

$$R_{xy}^{j}(k) = \frac{\sum_{i=1}^{N}(x_{k-i-j} - E[x_{k-j}])(y_{k-i} - E[y_k])}{(N-1)s_{x_{k-j}}s_{y_k}} \tag{5.1}$$

where x_{k-j} and y_k are the value of the variables at instant $k-j$ and k, respectively. $E[x_{k-j}]$ and $E[y_k]$ are sample means for the recent N (sample) values at instant $k-j$ and k, respectively, and $s_{x_{k-j}}$ and s_{y_k} are the sample standard deviations. Hence, $R_{xy}^{j}(k)$ describes the causal effect of x on y after j time units. The delay of the causal effect varies depending on the two variables that are monitored. For example, an increase in workload (e.g. the number of requests) may affect the CPU utilization almost immediately. In comparison, turning on a machine in a server cluster would not increase the total processing capacity instantly. To capture the causal relations more effectively, a number of different correlation values (in terms of delays) can be calculated. For example, AdaptGuard can try $R_{xy}^{1}(k)$ and $R_{xy}^{2}(k)$ to infer the causality assumption, $x \rightarrow y$. AdaptGuard then picks the one with the highest absolute value that shows the strongest correlation. Note that the best value for j in the $R_{xy}^{j}(k)$ depends also on the sampling frequency; a higher sampling frequency may lead to a higher j value.

Then, AdaptGuard maps the continuous correlation value, R_{xy}^{j} (with the highest absolute value) to a discrete set of causal relations, $\{+, -\}$. To do so, a cut-off threshold, τ, is used such that correlation values above τ are interpreted as positive, those below $-\tau$ as negative, and values in between as no correlation (no arc). The same procedures are applied to infer $y \rightarrow x$.

Observe that AdaptGuard in fact does not really develop a "quantitative" model to infer causality assumptions. It merely learns the sign of temporal correlation between a pair of variables. Differently from our approach, authors in [35] used a linear regression model to search relationships between system variables for fault detection and diagnosis. Also, authors in [23] presented an approach for capturing probabilistic relationships among system variables using a Gaussian mixture model for fault analysis. While those approaches may provide more precise and richer explanations, if the underlying model changes (e.g. due to workload changes), the inferred model (e.g. a model learned against one type of workload) would not hold any more, possibly giving false alarms in detecting faults. In comparison, our approach can still learn the sign correctly as long as the underlying causal relation remains unchanged, hence more robust to the changing operating environment.

Typically regulation policies need a small number of variables (usually less than 5 variables) to implement adaptation loops [1, 36]. Since AdaptGuard only focuses on the variables used in regulation policies, the number of the inferred causality assumptions that should be monitored tends to be small. Hence, in terms of scalability, AdaptGuard has a clear advantage over other fault detection mechanisms that explore all the possible variables to figure out variables of interest [23, 35, 70] or statistical or learning-based mechanisms that try to pin down the root cause of the problem [2, 18, 24, 68].

Adaptation Graphs

In order to reason about stability, it is useful to recap the notion of *adaptation graphs* that are presented in Section 2.3.1. Briefly, vertices in an adaptation graph represent system variables of interest (automatically identified by AdaptGuard as discussed above). In a particular implementation, these typically include key performance metrics (e.g. response time, utilization, etc) and performance control knobs (e.g., number of threads allocated to a task, CPU speed in a DVS-capable system, or the percentage of requests dropped by admission control). AdaptGuard does not understand the semantics of these variables. It merely sees the values communicated across the interfaces it provides to the programmer for sensor, actuator and regulation policy objects. The directed edges (arcs) in the adaptation graph represent the causality assumptions between such variables. The edges are labeled positive, "+", if the changes are in the same direction and negative, "-", if they are in opposite directions. These edges are automatically discovered by AdaptGuard by performing the aforementioned inference techniques. A cycle in the adaptation graph represents a corresponding adaptation loop and the sign of the loop is defined as the product of the signs of the arcs in the cycle. Since a stable feedback loop is negative, the sign of an adaptation loop should be maintained negative.

Detecting Assumption Violations

Once variables of interest are identified and correlations are discovered between these variables, an adaptation graph can be constructed. The automated-detection module monitors the sign of each arc in the adaptation graph to determine if a violation occurred. At each period, the automated-detection estimates the (current) sign of an arc $x \to y$ (in the same way when it constructs the adaptation graph) and compares it against the (previously learned) sign of the arc given in the adaptation graph. If they are different, there is a violation. Further, if the sign of the corresponding adaptation loop becomes positive, it indicates a positive feedback loop, meaning system instability is observed. This process of checking causality assumption violations and positive feedback loops can be efficiently done online, since it is performed on a small number of inferred assumptions and is not computationally complex.

Observe that the sample size N in Eq. (5.1) for calculating correlation values may affect both the stability and the responsiveness of the automated-detection, since we use the cut-off threshold to determine the sign of causality assumptions. If the sample size is too small and the calculated correlation values keep varying around the boundary of the threshold, it may create instabilities. In comparison, with a too large sample size, the automated-detection may not act promptly upon assumption violations. Further, it can incur more overhead.

Typically, the reason a correlation sign is flipped is because the designer's model (and hence variables measured) is inadequate. The actual system model has more inputs and some of these may not be measured. Notwithstanding this lack of observability, positive feedback is still detected from the measured loop sign and hence instability can be assumed even if measured performance variables look "normal".

While the broken assumptions can give us some hint why the loop is destabilized, it does not tell us what exactly caused the problem. However, the root-cause-agnostic attribute of AdaptGuard is desirable for online detection purposes, considering that finding out the root cause of the problem is time-consuming and usually not suitable for online detection purposes.

Observe that AdaptGuard will not catch all instability problems. In particular, unstable negative feedback is not detected. It is not equipped to detect unstable negative feedback because it only uses signs for stability analysis (and not loop gain). As long as the signs of all causality arcs remain the same, AdaptGuard will not intervene.

Recovery of Stability

When a violation is detected (i.e., an arc changes sign), the feedback control loops involving the bad arc are opened by AdaptGuard (which implements sensor and actuator communication as we described earlier), a backup control action is taken as defined *a priori* for each loop. In our current implementation, the loop is then resumed after a pause, in hope that the transient condition (e.g., thrashing) that caused the anomalous behavior may have passed. Figure 5.1 illustrates this procedure. The sequence of opening, fixing, and closing the loop is repeated with an exponential back-off in resuming the feedback. If the violation is transient, this approach usually works well as the feedback resumes correctly. If, after a pre-configured number of trials, the system cannot recover, feedback is not resumed. Since open-loop actions are stable, it is less likely that the execution of backup control actions severely disturbs the normal execution of the system.

The backup control action executed in the open-loop mode depends on the loop broken and must be specified in advance. Observe that, in the case of overload control loops in general, a safe open-loop action would be to reduce load using a non-regulation policy (e.g., drop all/most low priority requests). The backup control action may be different based on the main performance metric of applications. Suppose an adaptive system is managed by an energy-saving policy that uses dynamic voltage/frequency scaling. One can assume that power consumption is the main performance metric. Then, the backup control action is to set the voltage/frequency level to its lowest value in order to minimize energy consumption. If it is most important to process as many user requests as possible when violations occur, the backup control action should be to set the voltage/frequency level to its highest value.

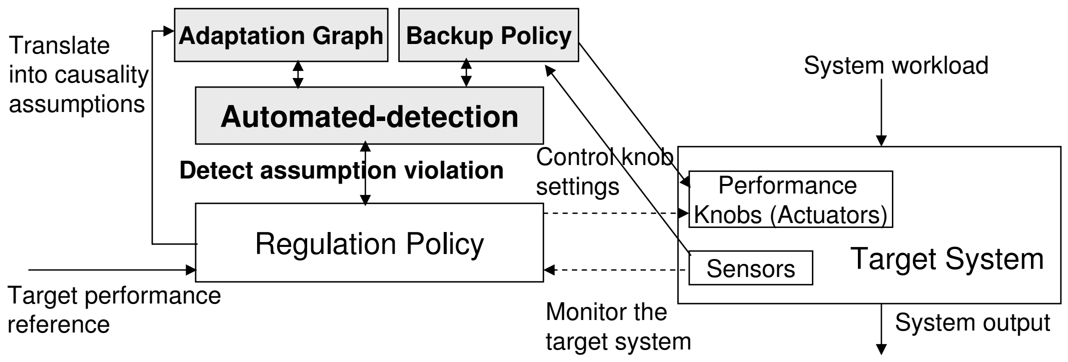

Translate into causality assumptions
Adaptation Graph
Backup Policy
Automated-detection
Detect assumption violation
Regulation Policy
Target performance reference
Control knob settings
Monitor the target system
System workload
Performance Knobs (Actuators)
Sensors
Target System
System output

Figure 5.1: The recovery action of AdaptGuard: when a violation is detected, a backup control action takes over the current regulation policy with the violation.

5.2.3 Evaluation

In this section, we thoroughly evaluate AdaptGuard using both artificial fault injection and a case-study on a QoS-adaptive Web server testbed. We first present the results of the fault injection followed by the results of the case study. Then we summarize them at the end of the section.

Fault Injection

In this section, we evaluate AdaptGuard by injecting various faults that cause causality assumption violations. First, we explain the application implementation as seen by AdaptGuard then proceed with a description of testbed set-up and the experiments performed.

1) Application Implementation:

We implement two typical regulation policies that are frequently found in the literature using AdaptGuard to manage the performance of a Apache Web server. The admission control policy [36] implements an admission control mechanism that adjusts the workload of the server by dropping a portion of incoming requests to prevent the system from being overloaded. The implemented control loop measures the current CPU utilization $Util$. The measurement is implemented as a sensor object that outputs $Util$ and sent to the admission control policy module that outputs a drop probability Pd to the actual admission controller (implemented as an actuator module), which drops a fraction Pd of incoming requests. Hence, the object outputs visible to AdaptGuard are $Util$ and Pd. AdaptGuard monitors correlations between these variables. Figure 5.2(a) shows the dominant causal relations. Indeed, as shown in figure, The arc $Util \rightarrow^{+}_{AC} Pd$ reflects the admission control policy output explaining that increased utilization, $Util$, will raise the drop probability, Pd, by the feedback loop. The arc $Pd \rightarrow^{-} Util$ illustrates that the utilization, $Util$, is enforced based on the drop probability, Pd. The two arcs form a cycle (a feedback loop) where the product of the signs of the arcs is negative, indicating a negative feedback loop.

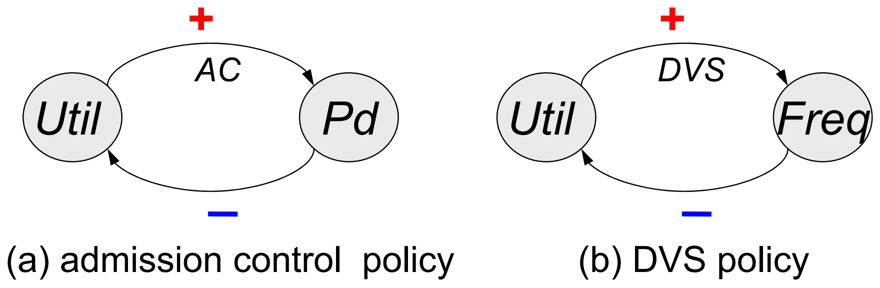


Figure 5.2: Adaptation graphs of the QoS-adaptive Web server (a) with the admission control policy and (b) with the DVS policy

The DVS policy [30] implements a power saving mechanism using dynamic voltage and frequency scaling (DVS). Similar to the admission control policy, the DVS policy measures the current CPU utilization $Util$. With the increased utilization beyond the target value, the DVS policy increases the frequency/voltage level $Freq$ to decrease the utilization and vice versa. Fig. 5.2(b) shows the dominant causal relations among them that form a negative feedback cycle.

2) Testbed Setup:

AdaptGuard is installed to monitor Apache Web server that runs on the same machine. The machine is equipped with an Intel Celeron 2.53GHZ CPU and 512MB of RAM. We instrument several sensors using AdaptGuard to collect measurements such as the inbound request rate (as seen by Apache) and CPU utilization. Actuators are implemented to enforce the decisions made by the two policies respectively. We use httperf to generate HTTP requests on a client machine with Intel Pentium IV 3GHZ CPU and 2GB of RAM. The inter-arrival time of the generated requests is exponentially distributed with a mean of 0.01 sec. The admission control policy tries to maintain the utilization around 0.45 and the DVS policy aims to maintain the utilization around 0.8. All machines are equipped with Redhat Fedora Core 4 Linux.

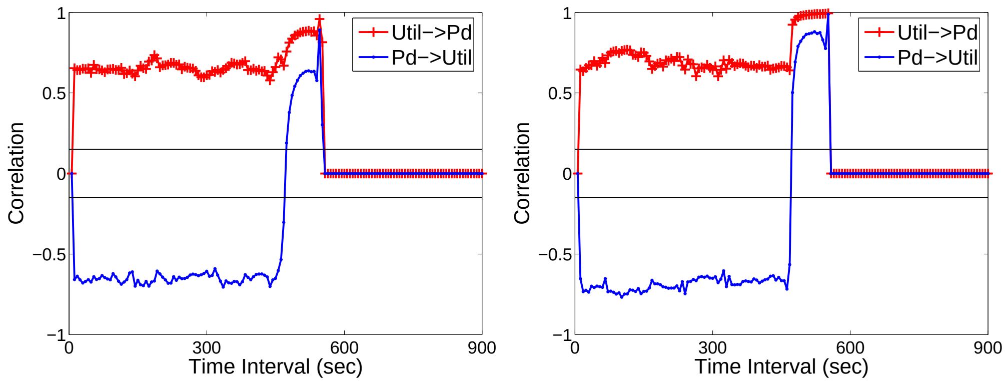
For the first set of experiments, we consider two different types of faults: the *busy loop* fault and the *missing file* fault. The missing file fault happens, for example, when an administrator mistakenly removes files from the HTML directory in a Web server. The busy loop fault happens when the exit condition of a loop is never met so that the program runs infinitely. The two faults are injected individually into the QoS-adaptive Web server that runs either the admission control policy or the DVS policy. No backup control actions are defined for the experiments since the purpose is to investigate if the AdaptGuard can successfully detect assumption violations caused by the injected faults.

For the second set of experiments, we investigate a more interesting scenario where the two regulation policies run at the same time in the QoS-adaptive Web server, causing the target system (the Web server) to destabilize due to conflicts between the two policies. This could happen even when there are no explicit software faults. In Chapter 2, we studied how one can detect such conflict at integration time. In this section, we show that AdaptGuard can detect causality violations caused by such conflict and restore acceptable performance at run-time. For comparison, we conduct experiments with and without the automated-detection. A backup control action is installed for the DVS policy that forces the system to operate at the maximum frequency for 30 seconds, assuming the main concern is the achieved throughput when violations occur. No backup control action is installed for the admission control policy.

We set the cutoff threshold τ to 0.15 to determine the sign of each edge from the correlation coefficient value and the sample size N is set to 50. The sampling frequency is set to 3 seconds.

3) Simple Fault Injection:

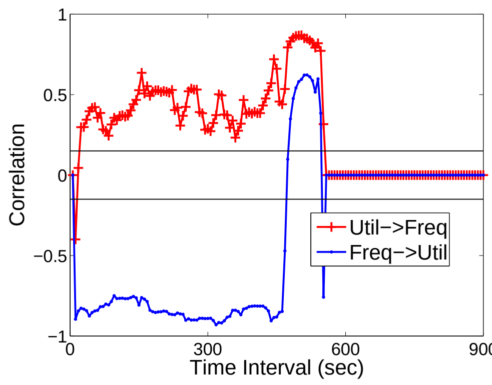
We present the results when the two faults, the missing file and the busy loop faults, are injected to the QoS-adaptive Web server with the DVS and the admission control policies. For each experiment, one of the two faults is injected into the Web server managed by one of the two policies. Hence, we conduct 4 experiments totally.



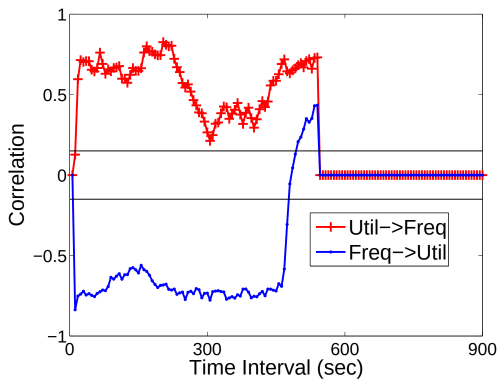
(a) Missing file: $Pd \rightarrow Util$ is violated (b) Busy loop: $Pd \rightarrow Util$ is violated

Figure 5.3: Faults are injected to a QoS-adaptive Web server with the admission control policy

Fig 5.3 depicts the result when the admission control policy runs and Fig 5.4 shows the result when the DVS policy runs. Faults are injected at 450th second and the experiments are conducted for total 900



(a) Missing file: $Freq \rightarrow Util$ is violated



(b) Busy loop: $Freq \rightarrow Util$ is violated

Figure 5.4: Faults are injected to a QoS-adaptive Web server with the DVS policy

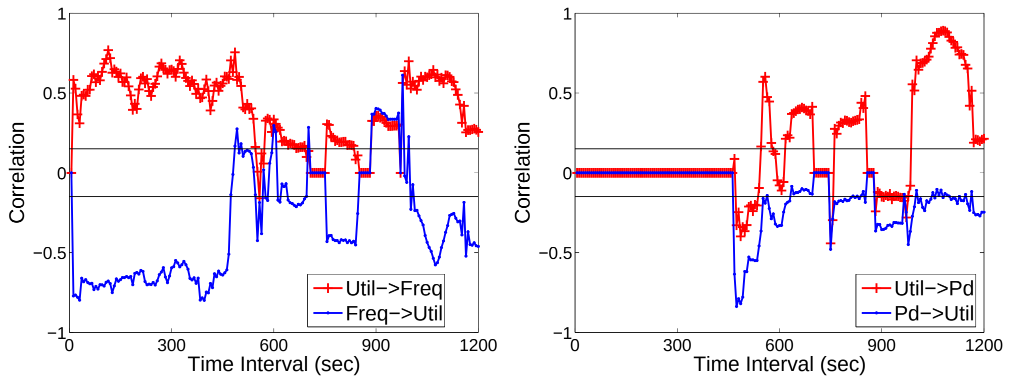
seconds. For all four cases shown in Fig 5.3 and Fig 5.4, AdaptGuard successfully detects violations as one of the two correlation coefficient values changes its sign shortly after the fault is injected. The missing file fault drives the system to experience low utilization regardless of the changes in the related variables (Pd for the admission control policy and $Freq$ for the DVS policy), as the requests for the missing files are rejected. Hence, the causality assumptions, $Pd \rightarrow^{-}_{AC} Util$ for the admission control policy and $Freq \rightarrow^{-}_{DVS} Util$ for the DVS policy, are violated. Similarly, the busy loop fault makes the causality assumptions, $Pd \rightarrow^{-}_{AC} Util$ for the admission control policy and $Freq \rightarrow^{-}_{DVS} Util$ for the DVS policy, broken as the utilization remains 100% once the fault is injected. A while after the faults are injected, all correlation values become *undefined value* (plotted as zero in the figures), since it is impossible to compute correlation values when one of the two variables does not change over time ($Util$ and $Freq$ remain unchanged).

Both types of faults are not easily recognizable without carefully investigating all the related variables. The difficult part is to differentiate the symptom from the normal behavior of the system. For example, with the busy loop fault, the 100% CPU utilization can be simply interpreted as the overload condition due to increased user requests. However, we demonstrated that AdaptGuard effectively detected the violations by monitoring only a small number of causality assumptions automatically identified.

4) Combined Regulation Policies:

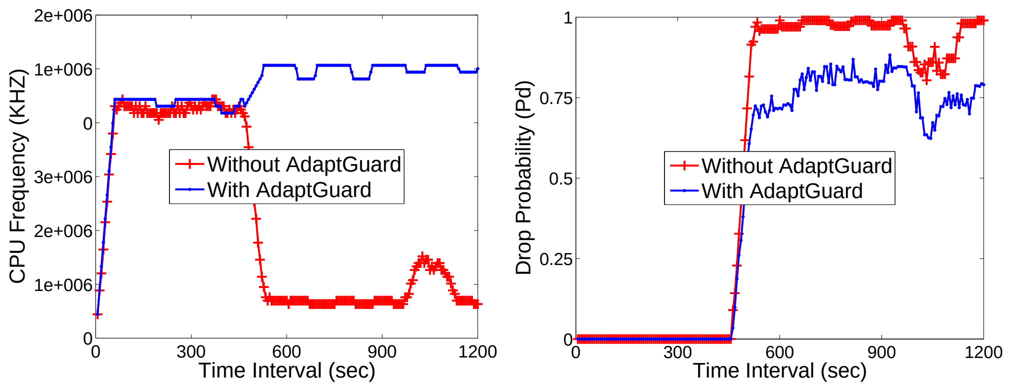
We present the results when the two (well-working) regulation policies are combined together. We first run the DVS policy alone for 450 seconds and start the admission policy at 450th second. Hence, after 450th second the two policies run together. The testbed previously explained is used.

Destabilization of the two regulation policies is clearly seen in Fig. 5.5. In Fig. 5.5(a), both of the two correlations of the DVS policy start to fluctuate after 450th second when the admission control policy is launched. Their sign changes unpredictably showing that the feedback loop implemented by the DVS policy destabilizes. The admission control policy also destabilizes as seen in Fig. 5.5(b). To help readers understand why this happens, we depict the values of the key variables such as the CPU frequency and the drop probability in Fig. 5.6. For comparison, we also show the values when the automated-detection module is present in the figure together.



(a) Correlations of the DVS policy (b) Correlations of the admission control policy

Figure 5.5: Correlations when the two regulation policies conflict and destabilize



(a) CPU frequency (b) Drop probability

Figure 5.6: Comparison of the key variables between when AdaptGuard is not used and AdaptGuard is used

In Fig. 5.6, without AdaptGuard, it is interesting to see that the frequency drops to the lowest level, while almost all incoming requests are dropped (the drop probability goes up to 1.0). This means that the application (Web server) does not accomplish any useful work. In detail, as the admission control policy starts to run, it sees that the utilization is around 80% since the DVS policy was already running. Hence, it drops requests to meet its target, 60%. As a result, the DVS policy gets to see the utilization is lower than its target, 80%, decreasing the frequency further to increase the utilization. This chain of actions forms a positive feedback cycle across the two policies, destabilizing the entire system to get all requests dropped eventually (see Fig. 5.6(b)), though the system is not really overloaded.

Merely monitoring performance metrics such as throughput, the CPU frequency, and the drop probability would not easily reveal the problem unless you have certain models established to compare with. AdaptGuard automatically learns the causality assumptions and detects their violations without any *a priori* model. When the assumption violations are detected in the DVS policy, the backup control action replaces the original DVS policy to set the CPU frequency to its maximum. As a result, it allows the CPU utilization to decrease, causing the admission control policy to drop less requests than the case without the automated-detection module.

Case-Study

In this section, we present a running case study to better demonstrate the efficacy and the limitations of our approach. We will demonstrate that AdaptGuard can successfully detect assumption violations caused by subtle interactions between a kernel-level mechanism for overload control and an admission control policy that manages a QoS-adaptive Web server. Further, we will show that AdaptGuard can effectively recover from performance degradation caused by the violations.

1) Testbed Setup:

In addition to CPU utilization $Util$, the admission control policy measures the current service request rate Req to adjust the drop probability Pd in a more fine-grained way. The drop probability Pd enforces the number of accepted requests Req which in turn affect the CPU utilization $Util$. Hence, the variables visible to AdaptGuard are Req, $Util$ and Pd. Figure 5.7 shows the identified causal relations. Further, the admission control policy serves client requests with multiple priority classes.

A QoS-adaptive Web server is implemented using AdaptGuard in the same way as presented in Section 5.2.3. In order to overload the Web server, we use total three client machines with Intel Pentium IV 3GHZ CPU and 2GB of RAM. Requests are generated using multiple instances of httperf such that the average request rate increases linearly over time to see reactions of the admission controller to the various

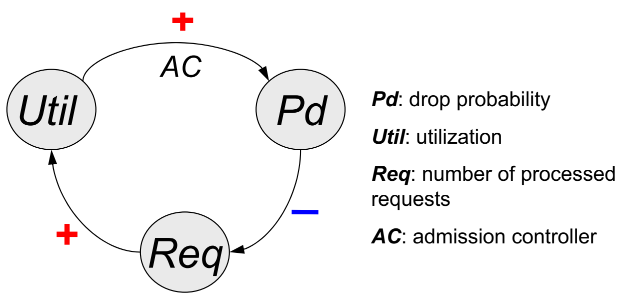
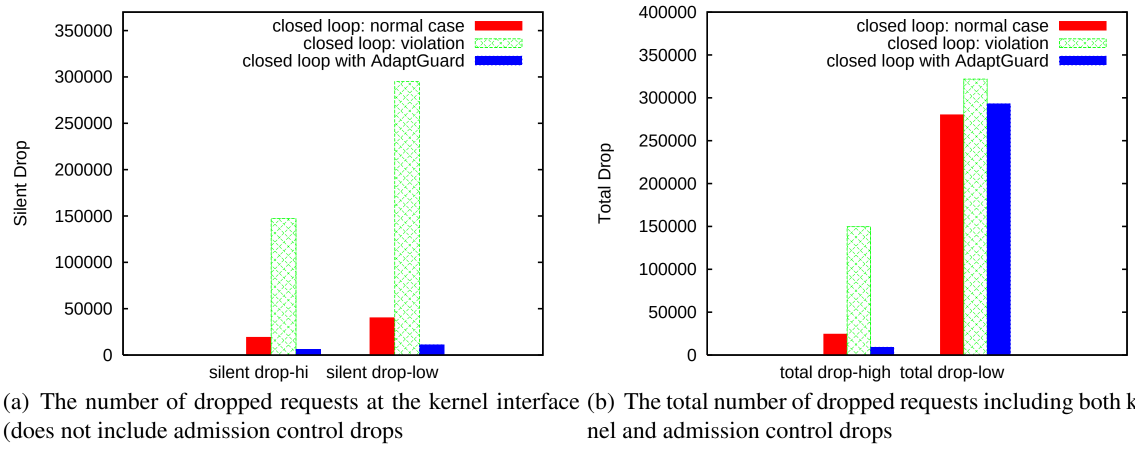


Figure 5.7: Adaptation graph of the QoS-adaptive Web server



(a) The number of dropped requests at the kernel interface (does not include admission control drops

(b) The total number of dropped requests including both kernel and admission control drops

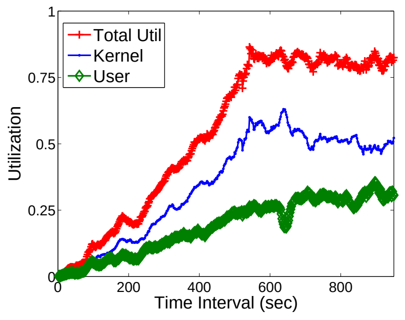
Figure 5.8: Comparison of the number of dropped requests

ranges of workload. The total request rate starts at 30 req/s and becomes 1500 req/s after 1000 sec. One client machine generates high priority class requests and the other two generate lower priority class requests.

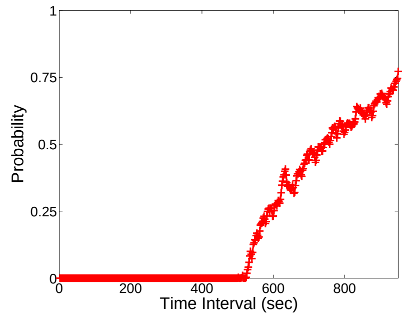
For comparison, we ran the system with and without the automated-detection module. Since the loop in our example is an overload control loop, backup control simply sets the drop probability Pd to a fixed high value for 30 seconds when the automated-detection module is present.

2) Experimental Results:

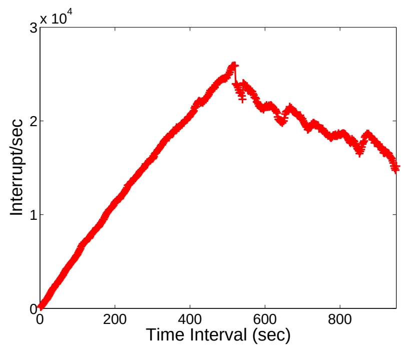
We observed that the closed loop admission control policy performs well most of time, protecting the system from being overloaded while keeping the utilization around the desired value of 0.8. Not all runs, however, are repeatable. On some occasions, the feedback loop is destabilized (at the same level of load that was previously handled successfully). A sharp indiscriminate increase is observed at the client side in dropped high-priority and low-priority requests alike. The high drop rate persists for both types of requests even though the external rate of high-priority requests alone should not overload the system. This is in direct contradiction to the design intent of the admission controller that is to drop low-priority requests first. As



(a) Utilization

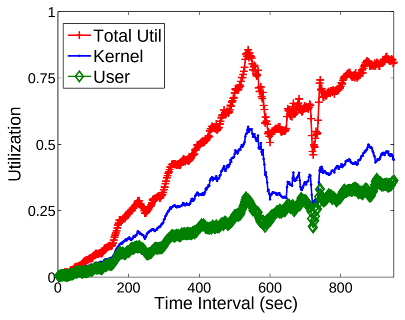


(b) Admission controller drop probability

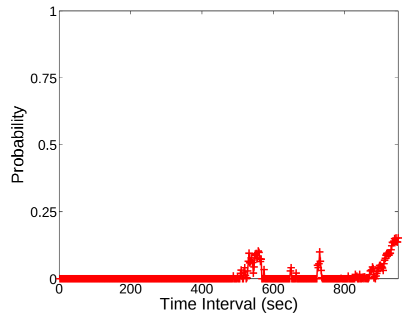


(c) Network interrupts

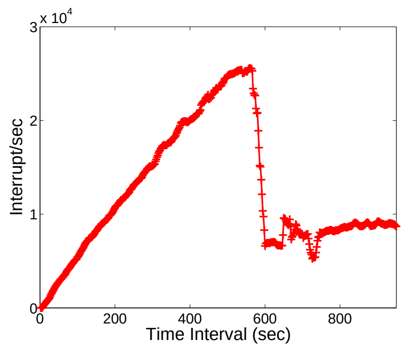
Figure 5.9: Results of closed loop when the admission controller works well



(a) Utilization



(b) Admission controller drop probability



(c) Network interrupts

Figure 5.10: Results of closed loop when the admission controller is destabilized

we show in the rest of the evaluation, the anomaly is due to an assumption violation triggered by the kernel's anti-livelock mechanism.

We then observed that when the automated-detection module is used, performance is quickly restored and the drop probability of higher-priority clients goes back to (approximately) zero. These results are shown in Figure 5.8(a) and Figure 5.8(b). Figure 5.8(a) shows the number of low-priority and high-priority requests dropped indiscriminately at the kernel interface when the feedback loop works well, when it does not work, and when AdaptGuard automated detection is used. Figure 5.8(b) shows the same for the total number of dropped requests as seen by the client (i.e., including both kernel and admission control drops).

The interesting part about this example, as we show shortly, is not that stability was restored. After all, AdaptGuard simply opens the unstable loop and we have only one feedback control loop in this simple example. The interesting part was that instability was detected at all. The issue is, to the *server*, all performance metrics looked normal while instability was in progress. Server utilization was moderate, the input request rate seen was low, the admission controller was fully open most of the time, and the delay was within bounds. The signs of the problem were manifest only at the client side, where most requests timed out. The disparity was because we did not have a sensor to measure drops from the kernel queue. The regular socket API does not offer such interface, and since server software is designed for portability it cannot depend on it. Drops from the kernel queue were substantial and completely invisible to the server. Nevertheless, AdaptGuard "conjectured" (given only those measurements that the designer used in their control loop) that the control loop must be unstable and disconnected it. In the following, we explain in more detail the above scenario and performance observations.

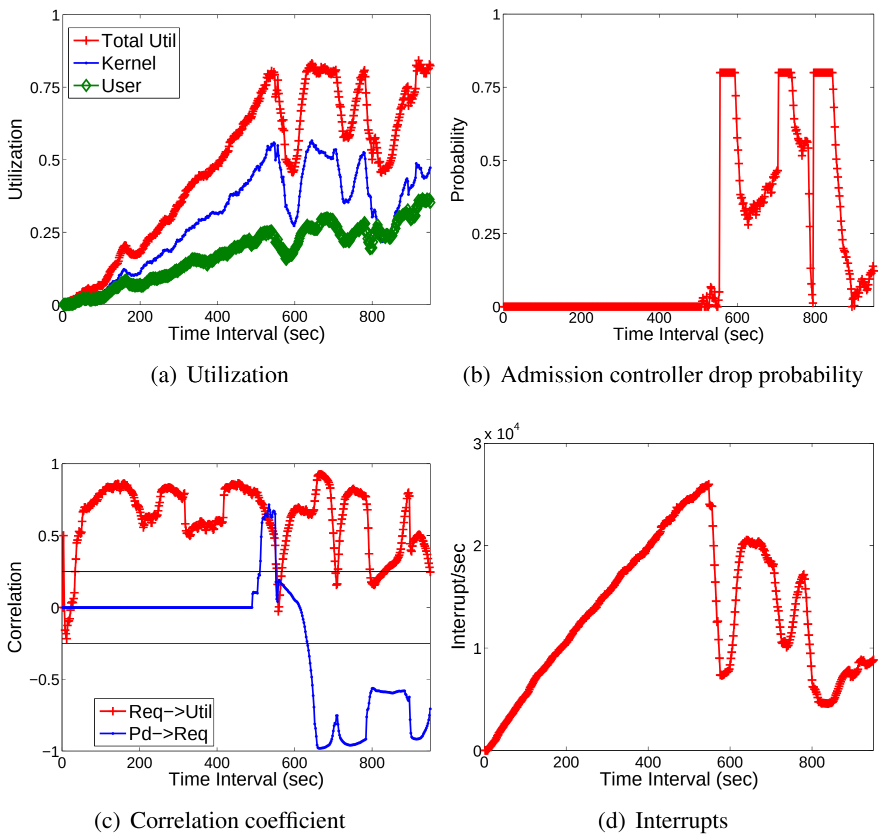
3) Closed Loop Control:

Figure 5.9 depicts closed loop performance when the admission controller works well. The utilization is successfully kept around the desired value of 0.8 (Figure 5.9(a)). As the incoming request rate increases, the admission controller increases the drop probability (Figure 5.9(b)) to keep the utilization around 0.8. Observe also that as the request rate increases the kernel overhead due to network interrupts increases as shown in Figure 5.9(c).

Figure 5.10 presents a case when the admission controller is destabilized. The figure shows that a sharp drop in utilization occurs (compare Figure 5.10(a) to Figure 5.9(a)) causing the admission controller to believe that the system is underloaded. Hence the admission controller stops dropping requests (compare

Figure 5.10(b) to Figure 5.9(b)). What the admission controller does not know (as shown in Figure 5.8(a)), is that a significant number of requests are now being dropped at the network card in the kernel. The utilization drop in Figure 5.10(a) was not due to a decrease in load. It is in fact triggered by an increase in load! The drop is attributed to the new Linux API called NAPI of kernel 2.6 that prevents livelocks by resorting to polling under heavy workload with reduced network interrupts. This results in an assumption violation since utilization decreased (instead of increasing) with increased request rate. The switch from interrupt-driven to mostly polling I/O is well illustrated in Figure 5.10(c), where we see a sharp drop in interrupt overhead.

While the admission controller believes that utilization has dropped and attempts to increase it, polling does not in fact catch all packets successfully, resulting in indiscriminate drops of packets at the network interface. Both high-priority and low-priority clients suffer as was shown in Figure 5.8(a) and Figure 5.8(b).



(a) Utilization (b) Admission controller drop probability

(c) Correlation coefficient (d) Interrupts

Figure 5.11: Closed loop with AdaptGuard

4) Closed Loop with AdaptGuard:

Figure 5.11 presents the result of running a closed loop experiment with AdaptGuard support for automated-detection. Figure 5.11(c) depicts the estimated correlation coefficient values of the arcs in the adaptation graph. We do not explicitly show the correlation coefficient for the arc $Util \rightarrow^{+} Pd$ since it is implemented by the admission control software, and thus does not constitute an interesting measurement. The correlation coefficient of arc $Pd \rightarrow Req$ becomes positive as the utilization drops sharply and stays negative after that, while that of $Req \rightarrow Util$ starts fluctuating after the load settles in. We set the threshold τ to 0.25. AdaptGuard checks causality assumption violation resolving the violation when detected. Note that the zero values (undefined values) of arc $Pd \rightarrow Req$ are not considered as violations since the controller was not in action. The resulting drop probability is depicted in Figure 5.11(b). Figure 5.11(a) depicts the utilization and Figure 5.11(d) shows network interrupts. The oscillations in the right half of those figures occurs as AdaptGuard invokes the backup action several times for a fixed amount of time.

The impact of AdaptGuard was illustrated in Figure 5.8(a) and 5.8(b). The third bar in each set shows the drops experienced with AdaptGuard. It can be seen from Figure 5.8(a) that the silent kernel drops are virtually eliminated. This explains the performance in Figure 5.8(b) showing that only low-priority requests are dropped while the high-priority requests are not, despite the assumption violation. Note also, that the mechanism does not increase the number of drops among low-priority clients. This demonstrates that AdaptGuard is effective at restoring proper function of the closed loop.

5.2.4 Conclusions

In summary, we developed and demonstrated, using a number of examples, AdaptGuard capable of detecting instability due to assumption violations. Such instability is insidious because it progresses while all performance measurements look "normal". In essence, AdaptGuard does not *see* the performance degradation either and hence does not *know* for a fact that the target system is unstable. This is because the target system does not offer access to the sensors needed to measure the performance degradation, and even if such measurements could be made, it is not clear in advance what sensors AdaptGuard should have to see *unexpected* problems. The contribution, therefore, lies in detecting the problem using only those sensors that the target system uses for their normal operation. Given only those sensors, AdaptGuard automatically "reverse-engineers" the causality assumptions the designer must have made, and monitors them for violations. When a violation is detected using those sensors, AdaptGuard "guesses" that the system may now be unstable (even though it cannot observe the instability), and takes action to restore stability successfully.

5.3 Diagnosis Using Discriminate Pattern Analysis

Since the diagnosis mechanism presented in the previous section uses statistical correlation to relate individual variables, it does not easily generalize to identifying sequences of discrete events that are correlated with problems. This is because individual events in the sequence may not be correlated with bad behavior, except when they occur in a particular order. Hence, techniques are needed that are order-sensitive. In this section we present a diagnosis mechanism that identifies chains of recurrent discrete events that may explain root causes of such problems.

In our previous work [30], we presented examples of the aforementioned interaction-induced performance problems. In one example, two independently working energy saving policies in a multi-tier web server farm interacted in a way that ended up *increasing* energy consumption. No components failed and the degraded performance (in terms of energy consumption) was not caused by bottlenecks. The key in diagnosing the root causes of such performance problems is to identify cyclic event patterns that can potentially explain the problem. Techniques based on classical control theory can identify vicious cycles (unstable loops) when the variables involved can be described by difference or differential equations, but when the cycles are composed in part of discrete events with no good models of the software systems that generate them, new different techniques are needed.

In order to identify cyclic patterns that cause performance degradation, we extend a data mining technique called *discriminative pattern analysis* that was successfully applied to identifying bad interaction patterns to diagnose protocol design bugs in wireless sensor networks [41, 42]. When the system performs poorly, the diagnosis module uses the extended algorithm to identify any anomalous chains of events consistent with "vicious cycles" that may explain the cause of the performance problem. To further reduce the number of such identified candidates and increase accuracy, it focuses on patterns that include semantically conflicting events. Semantically conflicting events are those that arise, for instance, when different performance management mechanisms make adjustments in conflicting directions, suggesting that they are "fighting" over the same actuation "knobs".

Given the traces of runtime events (defined by the user), the diagnosis module is invoked periodically and checks whether the system is working as expected or not. If the system is working as expected, the tool labels the collected trace for that time period as"good" and stores in the repository. If the systems performance was worse than expected, the trace is labled as "bad" and the diagnostic routine is launched. For example, in the server farm energy minimization example, when energy consumption significantly increases compared to the normal observation, collected events are labeled as “bad”. The discriminative analysis then identifies the culprit pattern that may be highly correlated to excessive energy spending.

AdaptGuard is deployed in a three tier web server testbed of 18 machines to evaluate our approach. To make the comparison of the scheme to prior works including the approach in the previous section more concrete, we choose to reproduce two real-life problem scenarios reported in earlier literature. In one case, we show how our tool successfully identifies a bad pattern that highlights a conflict between two independent energy saving policies, dynamic voltage scaling (DVS) policy and On/Off policy. In another case, the tool successfully attributes the cause of an anomalously low throughput to bad interactions between an admission controller and a dynamic voltage scaling (DVS) policy.

The rest of the section is organized as follows. In Section 5.3.1, we describe the overall design and implementation of the system along with the challenges and our solutions. Section 5.3.2 presents real-life case studies that show the effectiveness of our tool. Finally, Section 5.3.3 concludes the section.

5.3.1 Design

Briefly, the second diagnosis mechanism works as follows. During normal system operation, the diagnosis module collects traces of runtime events (defined by the user) from *event sensors*, and labels them as “good” logs. When system performance is degraded, the module labels the trace as “bad” and performs diagnosis to identify any anomalous sequences of events that are consistent with vicious cycles causing the performance problem. Performance degradation itself can either be flagged manually by a user of the diagnostic tool, or can be automatically identified by specifying limits of acceptable performance (e.g., delay < 3 sec).

To effectively identify repeated sequences of events, we extend data mining techniques called *discriminative pattern analysis*, previously applied to diagnosing bugs in wireless sensor networks [41, 42]. We omit the details of these techniques since data mining is not a contribution of this section. One should note that data mining is especially appropriate for diagnosing root causes of *non-reproducible* behavior in complex systems with a dynamic and time varying nature, because the observed behavior diversity itself enhances ability to learn [41]. Further, discriminative pattern analysis is adequate for an online service as it reduces the search space tremendously and hence the processing time for diagnosis.

Identifying culprit patterns can be perceived as a classification problem. The goal is to identify discriminative patterns (i.e., sequences of events) that can correctly separate the bad logs from the good logs. Observe that it is usually enough to log only basic actions of software components or policy modules that directly affect the performance of the system, because the purpose of our service is to find out what actions cause performance to deviate from the desired goal. For instance, if the problem lies in excessive energy consumption, it is sufficient to log primary actions of policies that directly affect power consumption (e.g., DVS increase/decrease and Machine On/Off operations).

The diagnosis module first generates frequent sequences of events in both good and bad logs. After frequent patterns are generated, a frequent pattern from the bad log is recognized as discriminative if the pattern is not found in the good log or has disproportionate support. If a pattern is identified as discriminative, it joins a set of candidate patterns to consider as possible root causes of the problem. It remains to rank-order the patterns according to the likelihood that they are responsible for a problem.

Our algorithm reports both cyclic and non-cyclic patterns. Since a vicious cycle is necessarily a cyclic pattern, discriminative cyclic patterns are given a higher ranking when patterns are reported to the user. A cycle of repeating events A, B, and C can be thought of as a set of attractor states that once entered, repeats indefinitely. Entry can occur at any event in the cycle. Hence, ABC, BCA, and CAB are the same. This equivalence is taken into account when counting frequent patterns. This reduces the number of reported patterns improving the usability of our approach.

We further develop a simple heuristic to reduce the number of false positives by focusing on patterns that are semantically conflicting. This requires user help with coloring events such that conflicts are identified based on color. For example, if the user is trying to determine the cause of excess energy consumption, the user can annotate actions that increase consumption (such as "TurnMachineOn" and "FrequencyIncrease") by the color *red* whereas those that decrease consumption (such as "TurnMachineOff" and "Frequency-Decrease") by color *green*. Our tool can then make more informed decisions regarding the importance of the patterns. Namely, if a pattern consists of events of color Green only, they can be safely ignored. On the contrary, if a discriminative pattern (i.e., one that does not normally occur) consists of events of both colors, it may be one that reflects a conflict among policies that needs to be reported. (Normal upwards and downwards adjustment of controls around a set point will also generate a mixed-color pattern, but it will not be discriminative since it occurs in normal operation as well.) Using this simple coloring scheme, our diagnostic service is able to prune uninteresting patterns and retain only those that are potentially revealing.

In general, a user may use N unique event colors, which leads to an N by N matrix, called the *conflict matrix*, where the (i, j)th element represents a *conflict value* between color i and color j. To properly compare patterns of different frequency and internal conflict values, pattern rankings are determined by the weighted sum of the frequency and the conflicts within the pattern, as follows:

$$\begin{aligned} ranking = \omega \cdot conflict\_value + (1 - \omega)\cdot \\ frequency/total\_frequency, \end{aligned} \tag{5.2}$$

where ω is a weight constant between 0 and 1. A large ω indicates that rankings are more weighted by conflict values. With the conflict matrix constructed, the diagnosis module can determine rankings of all identified frequent patterns. Note that, the conflict matrix is provided by the user and not intended to be exact. It is merely a means to express the user's subjective knowledge of compatibility between events. Not all events need to be colored. Those not colored can be thought of as being of a default color that has no conflicts with other colors.

We expect that in many cases, using two colors is sufficient. One color can be used to represent actions in a direction that should improve the performance metric of interest. Another can be used to represent actions in the opposite direction. Indeed, the evaluation section presents examples where only two colors are used.

Note that legacy systems are expected to evolve slowly over time. Various changes such as software and hardware upgrades, the set of tasks, system configurations etc. does not change drastically every day. This slow rate of evolution lets the user to build up a repository over long period of time that can represents the expected behavior of the system. Our diagnosis technique can leverage such knowledge to filter out unimportant patterns to improve the diagnosis accuracy for effective debugging.

5.3.2 Evaluation

To evaluate the effectiveness of our online diagnostic service, we reproduced two performance problems reported in the previous section and Chapter 2 and applied them to our tool. In both cases, we were able to successfully identify "vicious cycles", and hence the cause of the problem. We elaborate on the testbed setup and the case studies below.

Logged Events for Admission Controller	Color
DropProbabilityIncreased	Red
DropProbabilityDecreased	Green
Logged Events for dvs Policy	**Color**
FrequencyDecreased	Green
FrequencyIncreased	Red

Table 5.1: Logged Events for Case Study I

Logged Events for dvs Policy	Color
FrequencyDecreased	Green
FrequencyIncreased	Red
Logged Events for Machine On/Off Policy	**Color**
MachineTurnedOn	Red
MachineTurnedOff	Green

Table 5.2: Logged Events for Case Study II

Testbed Setup

We evaluated our tool on a three tier Web server farm that consists of 17 machines. Each machine has 2.53 GHz Intel Celeron Processor and 512MB of RAM. We used an extra machine with Pentium IV 3GHz CPU and 2 GB RAM to generate HTTP requests.

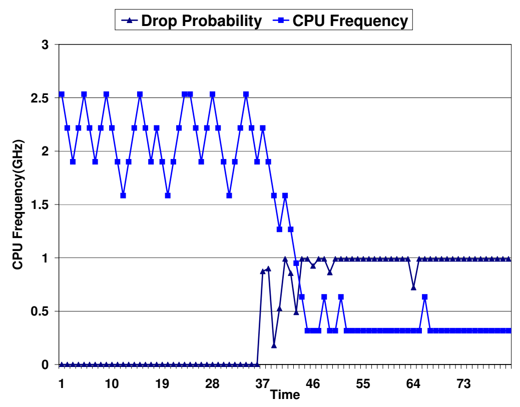
In the first case study, we use one machine (with 2.53 GHz CPU) to run a QoS-adaptive Web server. Another machine (with 3GHz CPU) generates HTTP requests to the Web server using *httperf*. The QoS-adaptive Web server aims to provide an acceptable quality of service to the user by keeping CPU utilization around the set point (thus not to overload the system). At the same time, it tries to save energy when the system is underutilized. We implement two software modules for this purpose: the *admission controller* and the *DVS policy*. The admission controller adjusts the probability to drop a client request dynamically based on the current CPU utilization (i.e., whether to drop a request or not) to keep utilization around the set point. Similarly, the DVS Policy decides whether to increase or decrease the CPU frequency based on the current CPU utilization to save energy when the system is underutilized. We choose to log the main operations of the two software modules: *DropProbabilityIncreased* and *DropProbabilityDecreased* of the admission controller that directly affect CPU utilization and *FrequencyDecreased* and *FrequencyIncreased* of the DVS policy that directly changes power consumption.

For the second case study, we configure a typical 3-tier Web server farm in our testbed where the first tier receives HTTP requests from the user, the second tier executes a business logic, and the third tier provides a persistent storage for the second tier. For this case study, all 17 machines are used: each tier is given 5 machines and two machines are used for load balancers for the first and third tier respectively. The extra machine with 3GHz CPU is used for generating HTTP requests. We use a well-known Web benchmark, TPC-W, to construct an Amazon-like 3-tier Web site. The performance goal is to save energy while response times are constrained to ensure a proper service quality for users. We enforced two energy saving policies: the DVS policy (explained above) and the *Machine On/Off policy* together. The Machine On/Off policy dynamically decides whether to turn on additional machine or to turn off machines depending on the average delay of the server farm. The same principle was applied for choosing which events to log. We log the main operations of the two policies that affect both power consumption and the delay: *FrequencyDecreased*, *FrequencyIncreased*, *MachineTurnedOn*, and *MachineTurnedOff*.

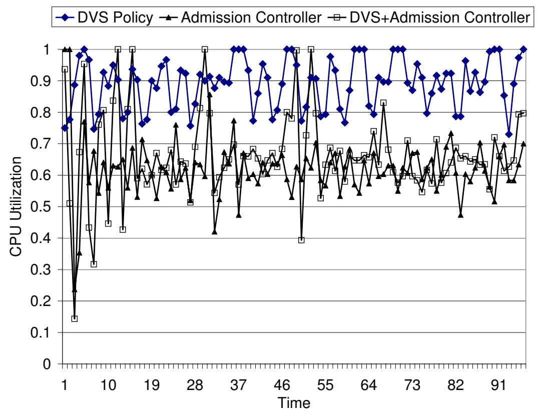
We label the logged events as either "Green" or "Red" depending on whether the corresponding decision is desirable or not from the perspective of the corresponding policies. For example, increasing the drop probability is not desirable and hence assigned the color Red. In contrary, decreasing CPU frequency is always desirable from the perspective of the energy saving policy and hence assigned the Green color. Logged events and the associated modules are listed in Table 5.1 and Table 5.2. We elaborate these experiments and the corresponding diagnosis below.

Case Study - I

In our first case study, we applied out tool to troubleshoot a performance problem in a QoS-adaptive Web server with the admission controller and the DVS policy installed. In short, the admission controller probabilistically decides whether to accept a client request or to drop based on the current CPU utilization. If the CPU utilization is higher than 60%, it drops a client request with high probability. The DVS policy periodically checks the CPU utilization and decreases the processor frequency to save energy if the current CPU utilization is lower than some predefined threshold (e.g., 90%). If the CPU utilization is higher than the threshold, it tries to increase the processor frequency. The admission controller is concerned with the quality of service and the DVS policy is an energy saving policy. Although the two components are trying to optimize two different metrics(i.e., average delay and energy), they interfere with each other and cause the system to perform poorly as shown in Figure 5.12(a) and in Figure 5.12(b).



(a) Conflict between DVS policy and the admission controller



(b) Effect on CPU utilization due to Conflict between DVS policy and the admission controller

Figure 5.12: Case Study - I

1) The Problem:

As can be seen in Figure 5.12(a), initially only the DVS policy was in action and the CPU frequency oscillates around 2GHz. After a while (i.e., around point 37), we introduced the admission controller as well. As it can be seen from the figure that the admission controller reacted immediately and increased the drop probability as the CPU utilization was higher than its predefined threshold (60%). As a result, the admission controller starts dropping requests. This eventually reduces the workload and decreases the CPU utilization. As the CPU utilization went down, the DVS policy assumed that the server is underutilized and correspondingly reduced the CPU frequency to save energy. At the lower frequency, the CPU utilization again goes high for the same amount of workload and the admission controller increased the drop probability again and so on. Ultimately, the drop probability stabilizes around 1 and the CPU frequency is set at the lowest possible speed and the system got stuck at that point. As can be seen in Figure 5.12(b), the DVS policy alone maintains the average cpu utilization around the set point(e.g., 90%) and the admission controller alone maintains the average cpu utilization around the set point (e.g., 60%). But when both of these policies are put together, the average cpu utilization oscillates around 60% with large oscillations due to instability. Interestingly, the situation is much worse than what Figure 5.12(b) suggests which someone may fail to realize by just looking at Figure 5.12(b). The Figure 5.12(b) shows that the two policies enforced together has almost similar utilization as the admission controller alone. But the figure fail to reveal that this utilization is achieved at a much lower CPU speed which can be seen from Figure 5.12(a).

Reported Patterns	Color of the Pattern
1. $< DropProbabilityIncreased >, < FrequencyDecreased >$	(Red, Green)
2. $< DropProbabilityDecreased >, < FrequencyIncreased >$	(Green, Red)
5. $< DropProbabilityIncreased >, < FrequencyIncreased >$	(Red,Red)

Table 5.3: Discriminative patterns due to conflicts between the DVS policy and the admission controller

2) Diagnosis:

In the repository, we have two traces of good cases. One trace corresponds to the log that was collected when the DVS policy was enforced alone during testing and performed as expected. The other trace corresponds to the log that was collected when the admission controller was enforced alone during testing and performed as expected. During runtime, the throughput found to be much less than expected (i.e., less than when any of the individual policy was acting alone) and was labeled as bad log.

To diagnose the bug, we applied our tool and it came up with the patterns shown in Table 5.3 as discriminative. The patterns clearly identify the problem in one step. The pattern $< DropProbabilityIncreased >$ $, < FrequencyDecreased >$ clearly shows a conflict as while admission controller is dropping requests, the DVS policy is trying to save energy. Saving energy at the expense of quality of service is never desirable. Interestingly, $< DropProbabilityDecreased >, < FrequencyIncreased >$ explains the normal operation where DVS policy is trying to cope up with the increased workload by increasing CPU speed . The last pattern ($< DropProbabilityIncreased >, < FrequencyIncreased >$)just shows what happens when the system is overloaded.

Case Study - II

In our second case study, we applied our tool to troubleshoot an excessive energy spending problem in a 3-tier Web server farm. All servers in the Web server farm are equipped with the DVS policy and the On/Off policy in an effort to reduce power consumption when the system is underloaded. In short, the DVS policy periodically checks the CPU utilization and decreases the processor frequency if the CPU utilization is lower than some predefined threshold (e.g., 90%). If the CPU utilization is higher than the threshold, it would try to increase the processor frequency. Machine On/Off policy tries to optimize the number of machines that are on at a particular time depending on the current system load. It periodically checks the average delay of the client requests. If the average delay is higher than a predefined threshold, it turns on additional machine, otherwise it turns off machines.

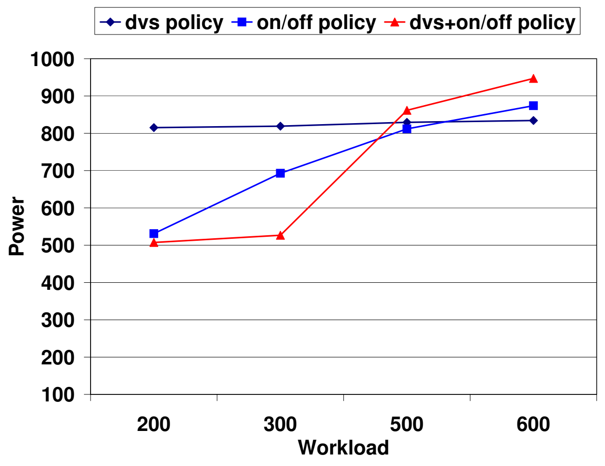


Figure 5.13: Applying dvs and on/off policy together consumes more energy at high load

Reported Patterns	Color of the Pattern
1. $< FrequencyDecreased >, < MachineTurnedOn >$	(Green,Red)
2. $< FrequencyIncreased >, < MachineTurnedOff >$	(Red,Green)
3. $< FrequencyIncreased >, < MachineTurnedOn >$	(Red,Red)

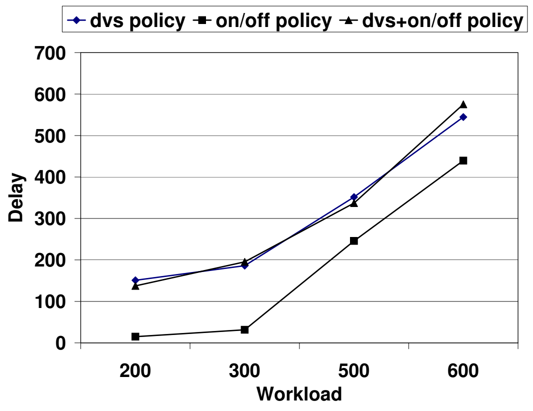
Table 5.4: Discriminative patterns due to conflict between the DVS policy and the On/Off policy

1) The Problem:

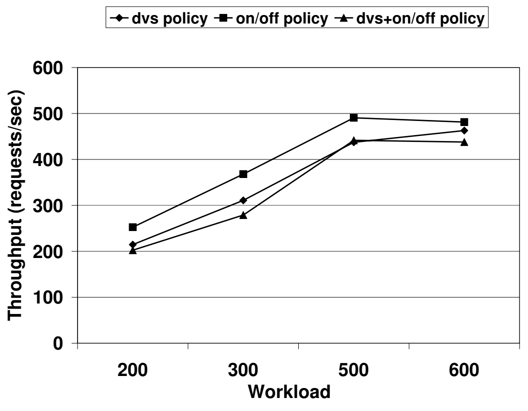
Interestingly, although the goal of both of the policies is to optimize energy consumption, when both policies were enforced together, the energy saving was less than when any one of the policies was enforced individually as shown in Figure 5.13. The reason was as follows. At high load, as the delay increases, the On/Off policy turned on additional machines. This eventually reduces the average workload per machine and decreases the CPU utilization. As the CPU utilization went down, the DVS policy assumed that the server is underutilized and correspondingly reduced the CPU frequency to save energy. At the lower frequency, the CPU utilization again goes high for the same amount of workload and the On/Off policy turned on more machines and so on. Ultimately, more machines were on at a lower frequency and overall energy consumption went high. Interestingly, the On/Off policy alone gives the best performance in terms of delay and throughput as shown in Figure 5.14(a) and Figure 5.14(b). This is due to the fact that the On/Off policy does not change the cpu speed and hence all the machines run at the highest speed.

2) Diagnosis:

In the repository, we have two traces of good cases. One trace corresponds to the log that was col-



(a) Effect on delay due to Conflict between DVS policy and the machine on/off policy



(b) Effect on throughput due to Conflict between DVS policy and the machine on/off policy

Figure 5.14: Case Study II

lected when the DVS policy was enforced alone during testing and performed as expected. The other trace corresponds to the log that was collected when the On/Off policy was enforced alone during testing and performed as expected. During runtime, at high load, the amount of energy consumed found to be higher than expected (i.e., higher than any of the individual policy alone) and was labeled as bad log. After analysis, the tool returned the patterns listed in Table 5.4 as discriminative. The pattern $< FfrequencyDecreased >$ $, < MachineTurnedOn >$ clearly suggests a conflict as the two policies are trying to do opposite (i.e., the DVS policy is trying to save energy by decreasing frequency while the machine on/off policy is turning on additional machine). This is counterintuitive as turning on additional machine while running other machines at low speed does not make any sense. This immediately explains the problem.

Interestingly, the pattern $< FrequencyIncreased >, < MachineTurnedOn >$ highlights the fact that the policy were trying to save overall energy by turning off machines and increasing frequency to offset for the reduced number of available machines. Although the pattern $< FrequencyIncreased >, < MachineTurnedOn >$ is not a conflicting pattern, we decided to retain such patterns as this highlights that the system was trying to cope up with high workload by increasing CPU speed and by turning on additional machines. These patterns are quite intuitive and help the developer to understand how the policies are actually affecting each other.

Algorithm Overhead

For the first case study, we collected traces for approximately 25 minutes. It took the discriminative pattern mining module less than one second to finish and to produce the patterns. For the second case study, we collected samples for approximately 15 minutes and the discriminative pattern mining module took less than one second to come up with the patterns. One thing to note is that the lengths of the discriminative patterns were two in our experiments as we logged events from only two system components for each experiment. The run time is expected to vary with the length of the discriminative patterns and the number of the system components involved in the analysis. Detail analysis of the performance of the data mining algorithm is outside the scope of our current work.

5.3.3 Conclusion

In this section, we extend AdaptGuard to perform diagnosis using a data mining technique, *discriminative pattern analysis*. The diagnosis technique in the previous section that used statistical correlation may work well where all variables are continuous and the specific order of events are not important to detect instability. On the contrary, the approach presented in this section is able to recognize discrete event patterns that represent the“vicious cycles” causing the problem. To reduce false positives, we developed a heuristic to discard patterns that are not semantically conflicting using a simple coloring scheme. We provided two real life case studies in a three tier Web server testbed of 17 machines to show that our tool can effectively diagnose the problems and provide more intuitive feedback to the administrator.

Chapter 6

Related Work

6.1 Performance-Sensitive and Adaptive Systems

A significant number of QoS adaptation and other performance adaptation policies have been reported in the computing literature over the last decade (e.g., [29, 61, 66, 67]). For example, several researchers used online feedback control to meet those goals [1, 36, 66, 67]. For example, Abdelzaher et al. [1] presented a Proportional-Integration (PI) control loop for an Apache Web server that adaptively changes the number of assigned processes (or threads) to meet soft real-time latency requirements. Admission control is another useful adaptive policy to adapt the system's response to the current workload. *Yaksha* uses admission probability to adjust load on a multi-tier Web server system to guarantee end-to-end delay requirements [36]. Admission probability is adjusted by a feedback control loop to meet the delay constraints. The authors of [21] present a soft real-time feedback control-based DVS policy combined with request batching for energy control. Similarly, a DVS policy is implemented in a stand-alone Apache Web server in [67] with multiple QoS service classes, which have soft real-time deadlines.

In much of the current literature, these policies are designed and evaluated in isolation, showing efficacy in achieving the performance requirements of the system. However, unintended interactions between independently designed adaptive policies have not traditionally been addressed. As these policies gain popularity in deployed systems, a significant number of applications, middleware components and operating system mechanisms will exhibit adaptive behavior. Future performance-sensitive and adaptive software will therefore likely include multiple adaptive components. For example, it might include an adaptive CPU allocation in the kernel scheduler, an adaptive admission controller at the application layer, and perhaps an adaptive energy management module in middleware. While these components will perform well in isolation, the interactions between them must be well understood to prevent unintended consequences. In this dissertation, we focus on such interactions between adaptive components in performance-sensitive systems.

6.2 Energy Minimization in Servers

Our choice of case study used in this dissertation is motivated by the importance of energy saving in multi-tier Web server farms. The cost of energy is one of their dominant operating costs. In large server farms, it is reported that 23-50% of the revenue is spent on energy [7, 21, 62]. This is because in order to handle peak load requirements, server farms are typically over-provisioned based on offline analysis. Thus, most of the time, energy-consuming idle resources exist in the system and considerable amounts of energy can be saved by reducing resource consumption during non-peak conditions.

Several previous works focus on DVS in stand-alone servers and server clusters. The authors of [21] present a soft real-time feedback control-based DVS policy combined with request batching. They show from simulation studies that up to 42% CPU energy is saved in a single Web server. They do not evaluate their method in a real system. A DVS policy is implemented in a stand-alone Apache Web server in [67] with multiple QoS service classes, which have soft real-time deadlines. Elnozahy et al. present and evaluate by simulation five different power management schemes for single-tier server clusters [20]. The schemes employ VOVO (vary-on/vary-off, i.e., turning nodes on and off depending on cluster load) and/or independent or coordinated (across the cluster) DVS. VOVO attempts to consolidate all workload to just as many nodes as necessary, leaving enough slack for load spikes. An independent DVS policy (IVS) is completely node-local, while a coordinated one (CVS) is constrained to a small frequency range around the cluster average. VOVO combined with CVS is shown to be superior. A theoretical framework to optimize power and performance at runtime for an e-business data center is derived in [44]. The authors evaluate power savings and performance of a multi-chip memory system server cluster. Their results demonstrate that around 70% savings are made in power as compared to static power management. Wang et al. [76] propose a fully decentralized control framework. In their work, the optimization problem is divided into sub-problems, each solved separately. Individual controllers are implemented using optimal control theory. The framework is applied to a computing cluster to minimize the power consumption while satisfying the specified response time requirement.

In this dissertation, we consider the problem of applying multiple power saving mechanisms (such as DVS, switching off machines, and admission control), and jointly optimizing them for performance subject to resource constraints. Further, our energy minimization application using OptiTuner is distinguished from the literature in the fact that we address the energy minimization problem with delay constraints in a multi-tier server farms evaluated in a real testbed of 17 machines.

6.3 Holistic Performance Management Approaches

Utility maximization has been widely adopted in many areas such as real-time systems and wireless sensor networks [30, 50, 53, 63]. Q-RAM [50] provides a centralized optimization solution to assign available system resources to multiple applications along multiple QoS dimensions in a way that maximizes the total utility. Q-RAM also presents fast approximate algorithms as an alternative to time-consuming optimal algorithms for NP-hard problems. The Time/Utility Function/Utility Accrual (TUF/UA) model [63] provides an effective way for resource management in time-critical applications. In this dissertation, we implement a centralized optimization approach using OptiTuner and show that it can effectively coordinate adjustment of multiple control knobs. We use an open source optimization solver, OpenOpt [82], to solve the energy minimization problem.

Utility maximization problems can be also solved in a distributed way under certain assumptions (e.g., convexity of problems). Recent breakthroughs in networking literature have led to the development of a mathematical theory for optimally layering network protocols using *optimization decomposition* techniques [10, 12, 15, 38, 69] to maximize the global utility of the involved network components. Optimization decomposition as applied to network layering has been summarized and explained in [12]. In this work, we show how optimization decomposition techniques can be applied to distributed performance optimization and resource management problems in performance-sensitive systems. Unlike centralized approaches, the distributed approach presented in this dissertation provides a distributed solution for achieving good aggregate performance. While focusing on a smaller set of problems assuming continuous variables and convexity of objective functions, however, we showed that the distributed approach presented in this dissertation works well in practical soft-real time systems.

MIMO control has been successfully used to maintain the desired performance in performance-sensitive systems. In [78] authors developed a two-layer control architecture using an MIMO feedback control approach to provide real-time guarantees for virtualized web servers while reducing power consumption. In their more recent work [79], they presented a power control algorithm for chip multiprocessors (CMPs) using MIMO optimal control theory to coordinate power consumptions of individual cores. While doing so, each core's temperature is kept below a certain threshold. In this dissertation, we implement an MIMO feedback algorithm used in [78] to show that OptiTuner can be easily used to apply an existing design technique. Further, through evaluation, we show that the presented MIMO control approach exhibits comparable performance to the two optimization approaches, successfully reducing undesirable interactions between different power saving mechanisms.

In [84], the authors designed and developed a middleware layer for QoS control, called ControlWare, that provides control-theoretic performance guarantees under uncertainty for Internet services. ControlWare specifically focuses on a control-theoretic approach to maintain performance around the desired set point. In comparison, OptiTuner is more general in the sense that it can support various holistic performance optimization and control techniques. Furthermore, ControlWare does not consider the aggregate behavior of controlling multiple performance knobs in the system, while OptiTuner's central goal is to ensure good aggregate behavior when multiple performance knobs are concurrently adjusted.

6.4 Problem Detection and Diagnosis in Software Systems

Several automated detection and diagnosis mechanisms have been proposed recently. Some of them have focused on troubleshooting problems caused by system misconfiguration [49, 75, 80]. For example, Wang et al. [75] proposed an automated troubleshooting approach to diagnose the root cause of system misconfiguration. It uses a statistical method to derive rankings on probable causes using empirical Bayesian estimation. On the other hand, several techniques tackled the isolation of performance problems or system failures. [2, 18, 24, 68]. For example, Aguilera et al. [2] proposed a tool that isolates the performance bottleneck in a distributed system composed of heterogeneous components (from different vendors without source code). It traces messages exchanged between the nodes to find a critical path that caused the system's latency. *Vertical profiling* [24] is a profiling technique to correlate various system measurements with each other to explain performance anomalies using statistical and visualization techniques.

All of the above debugging techniques focus on localizing the cause of performance problems or failures by finding faulty components (e.g., a faulty router), faulty code segments (e.g., a bad function), system misconfigurations, or bottleneck paths. Unfortunately, when *interactions* among individually well-behaved components are the problem, the conflict often cannot be easily localized. There are no malfunctioning components to identify and no bottleneck nodes to isolate in order to explain the performance problem. Our work provides tools to address this challenge.

Furthermore, all the above automated detection and diagnosis techniques are usually performed offline since they employ a complex statistical method. Our work complements those offline tools rather than replacing them, as it only provides a temporary recovery mechanism at run-time when causality assumptions are broken.

Run-time monitoring/verification techniques have been developed to provide assurance of the correctness of program execution [16, 25, 46]. In those systems, monitoring is usually performed based on a specification of system requirements (correct behaviors) and the target software system needs to be instrumented to notify the monitoring system of its state changes. For example, Java PathExplorer [25] monitors the execution of a Java program to check the system state against the desired system properties written in temporal logic provided by users. The Java bytecode is instrumented to provide an execution trace of the program, generating a sequence of events used by the monitoring process. AdaptGuard is similar, with the exception that causality assumptions to be monitored can be inferred by AdaptGuard in an automated fashion.

A number of approaches [23, 35, 48, 70] have been developed for the purpose of automatic fault detection and performance management without *a priori* knowledge or the help of domain experts, similar to our work. For example, Sun et al. [70] developed a mechanism for problem detection in distributed systems by constructing a state machine. However, it needs detailed system logs to build a state machine, while AdaptGuard does not need any to build adaptation graphs. Kumar et al. [48] developed an approach to automatically infer the relationship between the variables of interest and the controllable variables using probabilistic modeling techniques to derive component-level objectives. Although their approach can suggest new component-level objectives to meet the service level objectives (SLAs) according to changes in the operating environment, it cannot differentiate SLA violations from system instability.

In [23, 35], authors presented fault detection mechanisms using linear models [35] and probabilistic models [23]. These techniques aim to extract *invariants*, which are temporal correlations between a pair of system variables that hold all the time. By tracking abrupt changes in invariants, one can detect faults in distributed systems. However, they may trigger false alarms when the model is invalidated for benign changes, for example, such as user workload changes [22]. our work differs in that it does not use any quantitative model, only deriving the sign of temporal correlation. Hence, it is more robust to changes in the operating environment.

Finally, our work extends our previous efforts in the wireless sensor network community [40–42], where we presented tools to troubleshoot bugs in wireless sensor networks using frequent sequence mining. Our current work is more concerned with performance problems, and is more focused on finding self-reinforcing cycles that exacerbate such problems over time.

Chapter 7

Conclusions

As performance-adaptive systems become larger and their applications become more complicated, an increasing need arises for self-tuning and control components to accommodate environmental dynamics and uncertainty. A significant amount of recent work on server systems addressed automatic performance management. Such systems employ various adaptive components to tune different software performance knobs automatically to maintain acceptable performance and timing behavior in the face of a changing external environment.

However, without properly considering the combined effect of such adaptive components, self-reinforcing interactions between them may lead to bad states, significantly degrading performance. This dissertation focused on such problems of bad interactions between adaptive components. This work developed software support and design techniques to achieve performance composability in developing and running large-scale performance-adaptive systems.

We first presented a software service layer, *OptiTuner* that facilitates different holistic performance management techniques in distributed software. It provides proper abstractions and services to help the implementation of such approaches based on the concept of constrained optimization and control. Composition-time tools are provided by OptiTuner to uncover unsafe, unintended interactions between different adaptive software components. Further, an online diagnosis mechanism, called *AdaptGuard*, is presented to ensure system robustness with respect to residual errors at run-time. This improves robustness and achieve recovery in adaptive software systems.

Various realistic scenarios are implemented on a testbed comprised of 18 machines to evaluate the presented approaches. Recognizing the growing concern for the energy problem in data centers, we take as a main scenario an energy minimization application in a three-tier Web sever farm with multiple energy saving policies. OptiTuner is further evaluated in a consolidated environment using Xen virtual machines. Finally, a case study of a performance anomaly caused by unexpected interactions between an admission controller and the Linux anti-livelock mechanism is presented to evaluate AdaptGuard. Empirical results obtained from such scenarios demonstrate the efficacy of the approaches presented in this dissertation.

Much of our future infrastructure, such as power grids, healthcare, homeland defense systems, and disaster recovery systems will likely be vulnerable to performance problems investigated in this dissertation. We believe this work will lead to significant improvements in performance of adaptive distributed systems and has potential to result in significant pervasive impact on the cost and efficiency of our future software-driven infrastructure.

References

[1] Tarek F. Abdelzaher, Kang G. Shin, and Nina Bhatti. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Trans. Parallel Distrib. Syst.*, 13(1):80–96, 2002. ISSN 1045-9219. doi: http://dx.doi.org/10.1109/71.980028.

[2] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP '03*, pages 74–89, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-757-5. doi: http://doi.acm.org/10.1145/945445.945454.

[3] M. Allman, V. Paxson, and W. Stevens. TCP congestion control. RFC 2581, Internet Engineering Task Force, April 1999. URL `http://www.rfc-editor.org/rfc/rfc2581.txt`.

[4] Amazon.com. http://aws.amazon.com/ec2.

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5. doi: http://doi.acm.org/10.1145/945445.945462.

[6] Dimitri P Bertsekas. *Nonlinear programming*. Athena Scientific, 1995. ISBN 1886529140.

[7] Ricardo Bianchini and Ram Rajamony. Power and energy management for server systems. *Computer*, 37(11):68–74, 2004. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/MC.2004.217.

[8] J. V. BURKE, A. S. LEWIS, and M. L. OVERTON. The speed of shors r-algorithm. In *IMA Journal of Numerical Analysis*, 2008.

[9] Chi-Tsong Chen. *Linear System Theory and Design*. Oxford University Press, 1998.

[10] L. Chen, S. H. Low, and J. C. Doyle. Joint tcp congestion control and medium access control. In *IEEE Infocom*, volume 3, pages 2212–2222, 2005.

[11] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: problem determination in large, dynamic internet services. In *Intl. Conf. on Dependable Systems and Networks (DSN)*, pages 595–604, 2002.

[12] M. Chiang, S. H. Low, A. R. Calderbank, and J. C. Doyle. Layering as optimization decomposition: A mathematical theory of network architectures. *Proceedings of the IEEE*, 95(1):255–312, 2007. doi: 10.1109/JPROC.2006.887322. URL `http://dx.doi.org/10.1109/JPROC.2006.887322`.

[13] Ira Cohen, Moises Goldszmidt, Terence Kelly, Julie Symons, and Jeffrey S. Chase. Correlating instrumentation data to system states: a building block for automated diagnosis and control. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 16–16, Berkeley, CA, USA, 2004. USENIX Association.

[14] Continuent.org. The Sequoia Project. URL `http://sequoia.continuent.org/HomePage`.

[15] R. L. Cruz and A. Santhanam. Optimal routing, link scheduling, and power control in multihop wireless networks. In *IEEE Infocom*, volume 1, pages 702–711, 2003.

[16] Marcelo d'Amorim and Klaus Havelund. Event-based runtime verification of java programs. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, 2005. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/1082983.1083249.

[17] Yixin Diao, Neha G, Joseph L. Hellerstein, Sujay Parekh, and Dawn M. Tilbury. Using mimo feedback control to enforce policies for interrelated metrics with application to the apache web server. In *NOMS '02*, pages 219–234, 2002.

[18] Songyun Duan and Shivnath Babu. Guided problem diagnosis through active learning. In *ICAC '08*, pages 45–54, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3175-5. doi: http://dx.doi.org/10.1109/ICAC.2008.28.

[19] Electronic Educational Devices. Watts Up? Power Meter. URL `https://www.doubleed.com/secure.html`.

[20] E. N. Elnozahy, Michael Kistler, and Ramakrishnan Rajamony. Energy-efficient server clusters. In *PACS*, pages 179–196, 2002.

[21] E. N. Elnozahy, Michael Kistler, and Ramakrishnan Rajamony. Energy conservation policies for web servers. In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[22] Saeed Ghanbari and Cristiana Amza. Semantic-driven model composition for accurate anomaly diagnosis. In *ICAC '08*, pages 35–55, Washington, DC, USA, 2008. IEEE Computer Society. doi: http://dx.doi.org/10.1109/ICAC.2008.33.

[23] Zhen Guo, Guofei Jiang, Haifeng Chen, and Kenji Yoshihira. Tracking probabilistic correlation of monitoring data for fault detection in complex systems. In *DSN '06*, pages 259–268, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2607-1. doi: http://dx.doi.org/10.1109/DSN.2006.70.

[24] Matthias Hauswirth, Amer Diwan, Peter F. Sweeney, and Michael C. Mozer. Automating vertical profiling. In *OOPSLA '05*, pages 281–296, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: http://doi.acm.org/10.1145/1094811.1094834.

[25] Klaus Havelund and Grigore Roşu. An overview of the runtime verification tool java pathexplorer. *Form. Methods Syst. Des.*, 24(2):189–215, 2004. ISSN 0925-9856. doi: http://dx.doi.org/10.1023/B:FORM.0000017721.39909.4b.

[26] Jiayue He, Mung Chiang, and Jennifer Rexford. Tcp/ip interaction based on congestion price:stability and optimality. In *ICC '06*, 2006.

[27] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004. ISBN 047126637X.

[28] Jin Heo and Tarek Abdelzaher. Adaptguard: guarding adaptive systems from instability. In *ICAC '09*, pages 77–86, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-564-2. doi: http://doi.acm.org/10.1145/1555228.1555256.

[29] Jin Heo, Xue Liu, Lui Sha, and Tarek Abdelzaher. Autonomous delay regulation for multi-threaded internet servers. In *SPECTS 2006*, 2006.

[30] Jin Heo, Dan Henriksson, Xue Liu, and Tarek Abdelzaher. Integrating adaptive components: An emerging challenge in performance-adaptive systems and a server farm case-study. In *IEEE RTSS*, pages 227–238, 2007.

[31] Jin Heo, Praveen Jayachandran, Insik Shin, Dong Wang, and Tarek Abdelzaher. Optituner: An automatic distributed performance optimization service and a server farm application. In *FeBID 2009*, 2009.

[32] Jin Heo, Xiaoyun Zhu, Pradeep Padala, and Zhikui Wang. Memory overbooking and dynamic control of xen virtual machines in consolidated environments. In *IM '09*, pages 630–637, Piscataway, NJ, USA, 2009. IEEE Press. ISBN 978-1-4244-3486-2.

[33] Gavin Holland, Nitin Vaidya, and Paramvir Bahl. A rate-adaptive mac protocol for multi-hop wireless networks. In *MobiCom '01*, pages 236–251, New York, NY, USA, 2001. ACM Press. ISBN 1-58113-422-3. doi: http://doi.acm.org/10.1145/381677.381700.

[34] Tibor Horvath, Tarek Abdelzaher, Kevin Skadron, and Xue Liu. Dynamic voltage scaling in multitier web servers with end-to-end delay control. *IEEE Transactions on Computers*, 56(4):444–458, 2007. ISSN 0018-9340. doi: http://doi.ieeecomputersociety.org/10.1109/TC.2007.1003.

[35] Guofei Jiang, Haifeng Chen, and Yoshihira K. Discovering likely invariants of distributed transaction systems for autonomic system management. In *ICAC '06*, pages 199–208, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 1-4244-0175-5. doi: http://dx.doi.org/10.1109/ICAC.2006.1662399.

[36] Abhinav Kamra, Vishal Misra, and Erich M. Nahum. Yaksha: a self-tuning controller for managing the performance of 3-tiered web sites. In *IWQoS*, pages 47–56, 2004.

[37] Christopher Clark Keir, Christopher Clark, Keir Fraser, Steven H, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *In Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI*, pages 273–286, 2005.

[38] F. Kelly, A. Maulloo, and D. Tan. Rate control in communication networks: shadow prices, proportional fairness and stability. In *Journal of the Operational Research Society*, pages 237–252, 1998.

[39] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/MC.2003.1160055.

[40] Mohammad Maifi Hasan Khan, Tarek Abdelzaher, and Kamal Kant Gupta. Towards diagnostic simulation in sensor networks. In *Proceedings of the 4th DCOSS*, pages 252–265, 2008. Greece.

[41] Mohammad Maifi Hasan Khan, Hieu Khac Le, Hossein Ahmadi, Tarek F. Abdelzaher, and Jiawei Han. Dustminer: Troubleshooting interactive complexity bugs in sensor networks. In *Proceedings of the 6th SenSys*, pages 99–112, 2008. Raleigh, NC, USA.

[42] Mohammad Maifi Hasan Khan, Tarek Abdelzaher, Jiawei Han, and Hossein Ahmadi. Finding symbolic bug patterns in sensor networks. In *Proceedings of International Conference on Distributed Computing in Sensor Systems (DCOSS)*, California, USA, 2009.

[43] Gunjan Khanna, Ignacio Laguna, Fahad A. Arshad, and Saurabh Bagchi. Distributed diagnosis of failures in a three tier e-commerce system. In *26th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 185–198, Beijing, CHINA, October 2007.

[44] Bithika Khargharia, Salim Hariri, and Mazin S. Yousif. Autonomic power and performance management for computing systems. In *IEEE International Conference Autonomic Computing*. IEEE Computer Society, 2006.

[45] Martin G. Kienzle and K. C. Sevcik. Survey of analytic queueing network models of computer systems. *SIGMETRICS Perform. Eval. Rev.*, 8(3):113–129, 1979. ISSN 0163-5999. doi: http://doi.acm.org/10.1145/1009373.805453.

[46] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky. Formally specified monitoring of temporal properties. In *ECRTS '99*, 1999. URL `citeseer.ist.psu.edu/kim99formally.html`.

[47] Leonard Kleinrock. *Theory, Volume 1, Queueing Systems*. Wiley-Interscience, 1975. ISBN 0471491101.

[48] Vibhore Kumar, Karsten Schwan, Subu Iyer, Yuan Chen, and Akhil Sahai. A state-space approach to SLA based management. In *NOMS '08*, 2008.

[49] Franck Le, Sihyung Lee, Tina Wong, Hyong S. Kim, and Darrell Newcomb. Minerals: using data mining to detect router misconfigurations. In *MineNet '06*, pages 293–298, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-569-X. doi: http://doi.acm.org/10.1145/1162678.1162681.

[50] C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar, and J. Hansen. A scalable solution to the multi-resource qos problem. In *IEEE RTSS*, pages 315–326, 1999.

[51] Charles Lefurgy, Xiaorui Wang, and Malcolm Ware. Server-level power control. In *ICAC '07*, page 4, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2779-5. doi: http://dx.doi.org/10.1109/ICAC.2007.35.

[52] X. Lin, N. Shroff, and R. Srikant. A tutorial on cross-layer optimization in wireless networks. In *Selected Areas in Communications, IEEE Journal on*, volume 24, pages 1452–1463, 2006.

[53] Xue Liu, Qixin Wang, Wenbo He, Marco Caccamo, and Lui Sha. Optimal real-time sampling rate assignment for wireless sensor networks. In *ACM Transactions on Sensor Networks (TOSN)*, volume 2(2), pages 263–295, 2006.

[54] Zhuoqing Morley Mao, Ramesh Govindan, George Varghese, and Randy H. Katz. Route flap damping exacerbates internet routing convergence. *SIGCOMM Comput. Commun. Rev.*, 32(4):221–233, 2002. ISSN 0146-4833. doi: http://doi.acm.org/10.1145/964725.633047.

[55] Jeffrey C. Mogul. Emergent (mis)behavior vs. complex software systems. *SIGOPS Oper. Syst. Rev.*, 40(4):293–304, 2006. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/1218063.1217964.

[56] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Trans. Comput. Syst.*, 15(3):217–252, 1997. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/263326.263335.

[57] ObjectWeb Consortium. TPC-W benchmark based on the implementation from University of Wisconsin - Madison. URL `http://jmob.objectweb.org/tpcw.html`.

[58] Özgür Erçetin and Leandros Tassiulas. Market-based resource allocation for content delivery in the internet. *IEEE Transactions on Computers*, 52(12):1573–1585, 2003. ISSN 0018-9340. doi: http://dx.doi.org/10.1109/TC.2003.1252853.

[59] Pradeep Padala, Kang G. Shin, Xiaoyun Zhu Mustafa, Uysal Zhikui Wang, and Sharad Singhal Arif. Adaptive control of virtualized resources in utility computing environments. In *In Proceedings of the European Conference on Computer Systems*, pages 289–302, 2007.

[60] Venkatesh Pallipadi and Alexey Starikovskiy. The ondemand governor. In *Proc. of the Linux Symposium*, volume 2, 2006.

[61] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *SOSP*, pages 89–102, 2001.

[62] Ramya Raghavendra, Parthasarathy Ranganathan, Vanish Talwar, Zhikui Wang, and Xiaoyun Zhu. No ”power” struggles: coordinated multi-level power management for the data center. In *ACM ASPLOS XIII*, pages 48–59, 2008.

[63] Binoy Ravindran, E. Douglas Jensen, and Peng Li. On recent advances in time/utility function real-time scheduling and resource management. In *IEEE ISORC*, volume 0, pages 55–60, 2005. doi: http://doi.ieeecomputersociety.org/10.1109/ISORC.2005.39.

[64] Jerry Rolia, Ludmila Cherkasova, Martin Arlitt, and Artur Andrzejak. A capacity management service for resource pools. In *WOSP ’05: Proceedings of the 5th international workshop on Software and performance*, pages 229–237, New York, NY, USA, 2005. ACM. ISBN 1-59593-087-6. doi: http://doi.acm.org/10.1145/1071021.1071047.

[65] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. Beyond softnet. In *USENIX ALS ’01*, pages 165–172, Berkeley, CA, USA, 2001. USENIX Association.

[66] Lui Sha, Xue Liu, Ying Lu, and Tarek Abdelzaher. Queueing model based network server performance control. In *RTSS ’02: Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS’02)*, page 81, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1851-6.

[67] Vivek Sharma, Arun Thomas, Tarek Abdelzaher, Kevin Skadron, and Zhijian Lu. Power-aware qos management in web servers. In *IEEE RTSS*, pages 63–72, 2003.

[68] Kai Shen, Ming Zhong, and Chuanpeng Li. I/o system performance debugging using model-driven anomaly characterization. In *FAST’05*, pages 309–322, Berkeley, CA, USA, 2005. USENIX Association.

[69] Weihuan Shu, Xue Liu, Zonghua Gu, and Sathish Gopalakrishnan. Optimal sampling rate assignment with dynamic route selection for real-time wireless sensor networks. In *RTSS ’08*, 2008.

[70] Kewei Sun, Jie Qiu, Ying Li, Ying Chen, and Weixing Ji. A state machine approach for problem detection in large-scale distributed system. In *NOMS ’08*, pages 317–324, 2008.

[71] Transaction Processing Performance Council. TPC Benchmark W (Web Commerce). URL `http://www.tpc.org/tpcw`.

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

[73] Werner Vogels. Beyond server consolidation. *Queue*, 6(1):20–26, 2008. ISSN 1542-7730. doi: http://doi.acm.org/10.1145/1348583.1348590.

[74] Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/844128.844146.

[75] Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Automatic misconfiguration troubleshooting with peerpressure. In *OSDI'04*, pages 245–258, Berkeley, CA, USA, 2004. USENIX Association.

[76] M. Wang, N. Kandasamy, A. Guez, and M. Kam. Adaptive performance control of computing systems via distributed cooperative control: Application to power management in computing clusters. In *IEEE International Conference Autonomic Computing*. IEEE Computer Society, 2006.

[77] Weihong Wang and Baochun Li. Market-based self-optimization for autonomic service overlay networks. *IEEE JSAC*, 23:2320–2332, 2005.

[78] Yefu Wang, Xiaorui Wang, Ming Chen, and Xiaoyun Zhu. Power-efficient response time guarantees for virtualized enterprise servers. In *RTSS '08*, pages 303–312, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3477-0. doi: http://dx.doi.org/10.1109/RTSS.2008.20.

[79] Yefu Wang, Kai Ma, and Xiaorui Wang. Temperature-constrained power control for chip multiprocessors with online model estimation. *ISCA 2009*, 2009.

[80] Yi-Min Wang, Chad Verbowski, John Dunagan, Yu Chen, Helen J. Wang, Chun Yuan, and Zheng Zhang. Strider: A black-box, state-based approach to change and configuration management and support. In *LISA '03*, pages 159–172, Berkeley, CA, USA, 2003. USENIX Association.

[81] Zhikui Wang, Xiaoyun Zhu, and Sharad Singhal. Utilization and SLO-based control for dynamic sizing of resource partitions. In *16 th IFIP/IEEE Distributed Systems: Operations and Management*, pages 24–26, 2005.

[82] www.openopt.org. OpenOpt. URL `http://openopt.org`.

[83] Citrix XenServer. http://www.citrixxenserver.com.

[84] Ronghua Zhang, Chenyang Lu, Tarek Abdelzaher, and John A. Stankovic. Controlware: A middleware architecture for feedback control of software performance. In *IEEE ICDCS*, pages 301–310, 2002.

[85] Xiaoyun Zhu, Zhikui Wang, and Sharad Singhal. Utility-driven workload management using nested control design. In *American Control Conference*, 2006.

[86] Xiaoyun Zhu, Donald Young, Brian J. Watson, Zhikui Wang, Jerry Rolia, Sharad Singhal, Bret McKee, Chris Hyser, Daniel Gmach, Rob Gardner, Tom Christian, and Ludmila Cherkasova. 1000 islands: an integrated approach to resource management for virtualized data centers. *Cluster Computing*, 12(1): 45–57, 2009.